

Software Optimization Guide for AMD Family 10h and 12h Processors

Publication # **40546** Revision: **3.12**
Issue Date: **February 2011**

Advanced Micro Devices 

© 2006–2011 Advanced Micro Devices, Inc. All rights reserved.

The contents of this document are provided in connection with Advanced Micro Devices, Inc. (“AMD”) products. AMD makes no representations or warranties with respect to the accuracy or completeness of the contents of this publication and reserves the right to make changes to specifications and product descriptions at any time without notice. The information contained herein may be of a preliminary or advance nature and is subject to change without notice. No license, whether express, implied, arising by estoppel or otherwise, to any intellectual property rights is granted by this publication. Except as set forth in AMD’s Standard Terms and Conditions of Sale, AMD assumes no liability whatsoever, and disclaims any express or implied warranty, relating to its products including, but not limited to, the implied warranty of merchantability, fitness for a particular purpose, or infringement of any intellectual property right.

AMD’s products are not designed, intended, authorized or warranted for use as components in systems intended for surgical implant into the body, or in other applications intended to support or sustain life, or in any other application in which the failure of AMD’s product could create a situation where personal injury, death, or severe property or environmental damage may occur. AMD reserves the right to discontinue or make changes to its products at any time without notice.

Trademarks

AMD, the AMD Arrow logo, and combinations thereof, 3DNow!, AMD Virtualization and AMD-V are trademarks of Advanced Micro Devices, Inc.

HyperTransport is a licensed trademark of the HyperTransport Technology Consortium.

Linux is a registered trademark of Linus Torvald.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

MMX is a trademark of Intel Corporation.

PCI-X and PCI Express are registered trademarks of the PCI-Special Interest Group (PCI-SIG).

Solaris is a registered trademark of Sun Microsystems, Inc.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Contents

Revision History xvii
Chapter 1 Introduction 1
1.1 Intended Audience 1
1.2 Getting Started 1
1.3 Using This Guide 2
1.3.1 Special Information 3
1.3.2 Numbering Systems 3
1.3.3 Typographic Notation 4
1.4 Important New Terms 4
1.4.1 Multi-Core Processors 4
1.4.2 Primitive Operations 4
1.4.3 Internal Instruction Formats 4
1.4.4 Types of Instructions 5
1.5 Key Optimizations 6
1.5.1 Implementation Guideline 6
1.6 What's New on AMD Family 10h and 12h Processors 6
1.6.1 AMD Instruction Set Enhancements 7
1.6.2 Floating-Point Improvements 7
1.6.3 Load-Execute Instructions for Unaligned Data 8
1.6.4 Instruction Fetching Improvements 8
1.6.5 Instruction Decode and Floating-Point Pipe Improvements 9
1.6.6 Notable Performance Improvements 9
1.6.7 Large Page Support 11
1.6.8 AMD Virtualization™ Optimizations 11
Chapter 2 C and C++ Source-Level Optimizations 13
2.1 Declarations of Floating-Point Values 14
2.2 Using Arrays and Pointers 14

2.3 Unrolling Small Loops 17
2.4 Arrange Boolean Operands for Quick Expression Evaluation 18
2.5 Expression Order in Compound Branch Conditions 19
2.6 Long Logical Expressions in If Statements 20
2.7 Dynamic Memory Allocation Consideration 21
2.8 Unnecessary Store-to-Load Dependencies 21
2.9 Matching Store and Load Size 23
2.10 Use of Function Prototypes 25
2.11 Use of const Type Qualifier 26
2.12 Generic Loop Hoisting 26
2.13 Local Static Functions 28
2.14 Explicit Parallelism in Code 29
2.15 Extracting Common Subexpressions 30
2.16 Sorting and Padding C and C++ Structures 31
2.17 Replacing Integer Division with Multiplication 32
2.18 Frequently Dereferenced Pointer Arguments 33
2.19 32-Bit Integral Data Types 34
2.20 Sign of Integer Operands 35
2.21 Accelerating Floating-Point Division and Square Root 36
2.22 Speeding Up Branches Based on Comparisons Between Floats 38
2.23 Improving Performance in Linux® Libraries 40
2.24 Aligning Matrices 41
Chapter 3 General 64-Bit Optimizations 43
3.1 64-Bit Registers and Integer Arithmetic 43
3.2 Using 64-bit Arithmetic for Large-Integer Multiplication 45
3.3 128-Bit Media Instructions and Floating-Point Operations 49
3.4 32-Bit Legacy GPRs and Small Unsigned Integers 49
Chapter 4 Instruction-Decoding Optimizations 51
4.1 DirectPath Instructions 52
4.2 Load-Execute Instructions for Floating-Point or Integer Operands 53

4.2.1 Load-Execute Integer Instructions 53
4.2.2 Load-Execute SSE/SSE2/SSE3 Instructions with Floating-Point or Integer Operands 54
4.2.3 Load-Execute x87 Instructions with Integer Operands 55
4.3 Loop Iteration Boundaries 56
4.4 32/64-Bit vs. 16-Bit Forms of the LEA Instruction 56
4.5 Take Advantage of x86 and AMD64 Complex Addressing Modes 57
4.6 Short Instruction Encodings 58
4.7 Stack Operations 59
4.8 Partial-Register Writes 60
4.9 Using LEAVE for Function Epilogues 64
4.10 Alternatives to SHLD Instruction 66
4.11 8-Bit Sign-Extended Immediate Values 67
4.12 8-Bit Sign-Extended Displacements 68
4.13 Code Padding with Operand-Size Override and Multibyte NOP 68
Chapter 5 Cache and Memory Optimizations 71
5.1 Memory-Size Mismatches 71
5.2 Natural Alignment of Data Objects 73
5.3 Store-to-Load Forwarding Restrictions 74
5.4 Good Practices for Avoiding False Store-to-Load Forwarding 80
5.5 Prefetch and Streaming Instructions 81
5.6 Write-Combining 89
5.7 L1 Data Cache Bank Conflicts 90
5.8 Placing Code and Data in the Same 64-Byte Cache Line 91
5.9 Memory and String Routines 92
5.10 Stack Considerations 94
5.11 Cache Issues When Writing Instruction Bytes to Memory 95
5.12 Interleave Loads and Stores 96
5.13 Using 1-Gbyte Virtual Memory Pages 96
Chapter 6 Branch Optimizations 99

6.1 Branch Alignment and Density 99
6.2 Three-Byte Return-Immediate RET Instruction 100
6.3 Branches That Depend on Random Data 101
6.4 Pairing CALL and RETURN 103
6.5 Nonzero Code-Segment Base Values 104
6.6 Replacing Branches 104
6.7 Avoiding the LOOP Instruction 106
6.8 Far Control-Transfer Instructions 106
6.9 Branches Not-Taken Preferable to Branches Taken 107
Chapter 7 Scheduling Optimizations 109
7.1 Instruction Scheduling by Latency 109
7.2 Loop Unrolling 110
7.3 Inline Functions 113
7.4 Address-Generation Interlocks 115
7.5 MOVZX and MOVSX 116
7.6 Pointer Arithmetic in Loops 116
7.7 Pushing Memory Data Directly onto the Stack 118
Chapter 8 Integer Optimizations 121
8.1 Replacing Division with Multiplication 121
8.2 Alternative Code for Multiplying by a Constant 125
8.3 Repeated String Instructions 128
8.4 Using XOR to Clear Integer Registers 129
8.5 Efficient 64-Bit Integer Arithmetic in 32-Bit Mode 130
8.6 Derivation of Algorithm, Multiplier, and Shift Factor for Integer Division by Constants 137
8.7 Optimizing Integer Division 143
8.8 Efficient Implementation of Population Count and Leading-Zero Count 144
Chapter 9 Optimizing with SIMD Instructions 147
9.1 Ensure All Packed Floating-Point Data are Aligned 148
9.2 Explicit Load Instructions 148

9.3 Unaligned and Aligned Data Access 149
9.4 Moving Data Between General-Purpose and MMX™ or XMM Registers 149
9.5 Use SSE Instructions to Construct Fast Block-Copy Routines in 32-Bit Mode or 64-bit Mode 150
9.6 EMMS Usage 151
9.7 Using SIMD Instructions for Fast Square Roots and Divisions 152
9.8 Use XOR Operations to Negate Operands of SSEx Instructions 154
9.9 Clearing MMX™ and XMM Registers with XOR Instructions 155
9.10 Finding the Floating-Point Absolute Value of Operands of SSE and SSE2 Instructions 156
9.11 Accumulating Single-Precision Floating-Point Numbers Using SSE and SSE2 Instructions 157
9.12 Complex-Number Arithmetic Using SSE, SSE2, and SSE3 Instructions 158
9.13 Optimized 4 X 4 Matrix Multiplication on 4 X 1 Column Vector Routines 164
9.14 Floating-Point-to-Integer Conversion 167
9.15 Reuse of Dead Registers 167
9.16 Floating-Point Scalar Conversions 168
9.17 Move/Compute Optimization 170
9.18 Scalar Merge Optimization 172

Chapter 10 x87 Floating-Point Optimizations 173

10.1 Using Multiplication Rather Than Division 173
10.2 Achieving Two Floating-Point Operations per Clock Cycle 174
10.3 Floating-Point Compare Instructions 178
10.4 Using the FXCH Instruction Rather Than FST/FLD Pairs 178
10.5 Floating-Point Subexpression Elimination 179
10.6 Accumulating Precision-Sensitive Quantities in x87 Registers 180
10.7 Avoiding Extended-Precision Data 181

Chapter 11 Multiprocessor Considerations 183

11.1 ccNUMA Optimizations 183
11.2 Writing Instruction Bytes to Memory on Multiprocessor Systems 193
11.3 Multithreading 194

11.4 Memory Barrier Operations 200
11.5 Optimizing Inter-Core Data Transfer 203
Chapter 12 Optimizing Secure Virtual Machines 209
12.1 Use Nested Paging 210
12.2 VMCB.G_PAT Configuration 211
12.3 State Swapping 211
12.4 Economizing Interceptions 212
12.5 Nested Page Size 213
12.6 Shadow Page Size 214
12.7 Setting VMCB.TLB_Control 214
12.8 TLB Flushes in Shadow Paging 215
12.9 Use of Virtual Interrupt VMCB Field 216
12.10 Avoid Instruction Fetch for Intercepted (REP) OUTS Instructions 217
12.11 Share IOIO and MSR Protection Maps 219
12.12 Obey CPUID Results 219
12.13 Using Time Sources 220
12.14 Paravirtualized Resources 221
Appendix A Microarchitecture of AMD Family 10h and 12h Processors 223
A.1 Key Microarchitecture Features 224
A.2 Microarchitecture of AMD Family 10h and 12h Processors 225
A.3 Superscalar Processor 225
A.4 Processor Block Diagram 225
A.5 Processor Cache Operations 228
A.5.1 L1 Instruction Cache 228
A.5.2 L1 Data Cache 228
A.5.3 L2 Cache 228
A.5.4 L3 Cache 229
A.6 Branch-Prediction Table 229
A.7 Fetch-Decode Unit 230
A.8 Sideband Stack Optimizer 230

A.9 Instruction Control Unit 230
A.10 Translation-Lookaside Buffer 230
A.10.1 L1 Instruction TLB Specifications 231
A.10.2 L1 Data TLB Specifications 231
A.10.3 L2 Instruction TLB Specifications 231
A.10.4 L2 Data TLB Specifications 231
A.11 Integer Unit 231
A.11.1 Integer Scheduler 231
A.11.2 Integer Execution Unit 231
A.12 Floating-Point Unit 233
A.12.1 Floating-Point Scheduler 233
A.12.2 Floating-Point Execution Unit 234
A.13 Load-Store Unit 234
A.14 Write Combining 235
A.15 Integrated Memory Controller 235
A.16 HyperTransport™ Technology Interface (AMD Family 10h only) 236
A.16.1 HyperTransport Assist (AMD Family 10h only) 237

Appendix B Implementation of Write-Combining 239

B.1 Write-Combining Definitions and Abbreviations 239
B.2 Programming Details 240
B.3 Write-Combining Operations 240
B.4 Sending Write-Buffer Data to the System 241
B.5 Write Combining to MMI/O Devices that Support Write Chaining 241

Appendix C Instruction Latencies 245

C.1 Understanding Instruction Entries 246
C.2 General Purpose and Integer Instruction Latencies 250
C.3 System Instruction Latencies 260
C.4 128-Bit Media Instruction Latencies 265
C.5 64-Bit Media Instruction Latencies 279

C.6 x87 Floating-Point Instruction Latencies 284

Appendix D Tools and APIs for AMD Family 10h and 12h ccNUMA Multiprocessor Systems 289

D.1.1 Support Under Linux® 289

D.1.2 Support under Solaris™ 290

D.1.3 Support under Microsoft® Windows® 291

D.2 Tools and APIs for Node Interleaving 291

D.2.1 Support under Linux® 291

D.2.2 Support under Solaris™ 292

D.2.3 Support under Microsoft® Windows® 292

D.2.4 Node Interleaving Configuration in the BIOS 292

Appendix E NUMA Optimizations for I/O Devices 293

E.1 AMD64 System Topologies 293

E.2 Key Optimizations 293

E.3 Determining Number Of Nodes in AMD Family 10h and 12h Processor Systems in User-Mode 295

E.4 Allocating I/O Device Buffers 295

E.5 I/O Thread Scheduling 296

E.6 Using Write-Only Buffers for Device Consumption 296

E.7 Access of PCI Configuration Register 297

E.8 Determine the Nodes that Have Noncoherent HyperTransport™ I/O Links 299

E.9 Using Interrupt Affinity 304

Appendix F Remarks on the RDTSC(P) Instruction 305

Appendix G Guide to Instruction-Based Sampling on AMD Family 10h and 12h Processors 307

G.1 Background 307

G.2 Overview 308

G.3 IBS fetch sampling 309

G.3.1 Taking an IBS fetch sample 309

G.3.2 Interpreting IBS fetch data 310

G.4 IBS op sampling 312

G.4.1 Taking an IBS op sample 312
G.4.2 Interpreting IBS op data 313
G.4.3 Interpreting IBS branch/return/resync op data 315
G.4.4 Interpreting IBS Load/Store Data 316
G.4.5 Interpreting IBS load/store Northbridge data 317
G.5 Software-based analysis 319
G.5.1 Derived events and post-processing 320
G.5.2 Derived events for IBS fetch data 322
G.5.3 Derived Events for all Ops 323
G.5.4 Derived events for IBS branch/return/resync ops 323
G.5.5 Derived events for IBS load/store operations 324
G.6 Derived Events for Northbridge Activity 325

Figures

Figure 1. Memory-Limited Code 87
Figure 2. Processor-Limited Code 88
Figure 3. Simple SMP Block Diagram....184
Figure 4. AMD Family 10h 2P System....185
Figure 5. Dual Quad-Core AMD Family 10h Processor Configuration 185
Figure 6. Block Diagram of a ccNUMA AMD Family 10h Quad-Core Multiprocessor System 186
Figure 7. Internal Resources Associated with a Multiprocessor Node 187
Figure 8. AMD Family 10h Processors Block Diagram 226
Figure 9. AMD Family 12h Processors Block Diagram 227
Figure 10. Integer Execution Pipeline....232
Figure 11. Floating-Point Unit 234
Figure 12. Load-Store Unit 235
Figure 13. Configuration Address Register (0CF8h) 297
Figure 14. Configuration Data Register (0CFCh) 298
Figure 15. Link Type Registers F0x[F8, D8, B8, 98] 300
Figure 16. MMIO Base Address Registers F1x(B8h, B0h, A8h, A0h, 98h, 90h, 88h, 80h) 301
Figure 17. MMIO Limit Address Registers F1x[BC, B4, AC, A4, 9C, 94, 8C, 84]....301
Figure 18. Configuration Map Registers F1x[E0h, E4h, E8h, ECh] 303
Figure 19. Histogram for the IBS Fetch Completed Derived Event 321

Tables

Table 1. Instructions, Macro-ops and Micro-ops 5
Table 2. Optimizations by Rank....6
Table 3. Comparisons Against Zero....39
Table 4. Comparisons Against Positive Constant 39
Table 5. Comparisons Among Two Floats....39
Table 6. Prefetching Guidelines 83
Table 7. DIV/IDIV Latencies....144
Table 8. Integer Hardware Divider Latency (12H only)....144
Table 9. Single-Precision Floating-Point Scalar Conversion....168
Table 10. Double-Precision Floating-Point Scalar Conversion 169
Table 11. Write-Combining Completion Events....240
Table 12. Latency Formats....249
Table 13. General Purpose and Integer Instruction Latencies 250
Table 14. System Instruction Latencies 261
Table 15. 128-Bit Media Instruction Latencies....265
Table 16. 64-Bit Media Instruction Latencies....280
Table 17. x87 Floating-Point Instruction Latencies 285
Table 18. Size of Base Address Register 301
Table 19. IBS Hardware Event Flags....310
Table 20. Event Flag Combinations....311
Table 21. IbsOpData MSR Event Flags and Counts....315
Table 22. Execution Status Indicated by IbsOpBrnMisp and IbsOpBrnTaken Flags....315
Table 23. Execution Status Indicated by IbsOpReturn and IbsOpMispReturn Flags....316
Table 24. IbsOpData3 Register Information 317
Table 25. IbsOpData2 Register Fields 318
Table 26. Northbridge Request Data Source Field 319
Table 27. IBS Northbridge Event Data 319
Table 28. An IBS Fetch Sample....320
Table 29. 2-D Table of IBS Fetch Samples 320
Table 30. New Events Derived from Combined Event Flags 322
Table 31. Derived Events for All Ops....323
Table 32. Derived Events to Measure Branch, Return and Resync Ops....323
Table 33. Derived Events for Ops That Perform Load and/or Store Operations....324
Table 34. IBS Northbridge Derived Events 325

Revision History

Date	Rev.	Description
November 2010	3.12	Added information specific to Family 12h Updated Chapters 8, 9 and 11 and Appendices. Updated other chapters with change from mention of AMD Family 10h Processors to AMD Family 10h and 12h Processors; as appropriate.
May 2009	3.11	Added 9.17, 'Move/Compute Optimization' on page 170. Amplified A.15, 'Integrated Memory Controller' on page 235. Added A.16.1, 'HyperTransport Assist (AMD Family 10h only)' on page 237. Updated latencies for CPUID and INVPLG instructions in Appendix C.
February 2009	3.10	Corrected examples in 'Address-Generation Interlocks'. Clarified 'Write-Combining Operations'. Corrected and clarified 'HyperTransport™ Tunnels and Write Chaining (AMD Family 10h only)'. Added 'VectorPath Instruction Latencies.'
November 2008	3.09	Corrected the document title.
October 2008	3.08	Corrected assemby code examples in section 4.5, 'Take Advantage of x86 and AMD64 Complex Addressing Modes' on page 57. Made minor clarifications in 4.10, 'Alternatives to SHLD Instruction' on page 66. Clarified latency information in 6.7, 'Avoiding the LOOP Instruction' on page 106. Corrected assembly language statements in 7.2, 'Loop Unrolling' on page 110. Clarified latency discrepancies between processor revisions in 'VectorPath Instruction Latencies' on page 249.
April 2008	3.06	Revised section 4.13, on page 'Code Padding with Operand-Size Override and Multibyte NOP' on page 68. Revised section 9.16, 'Floating-Point Scalar Conversions' on page 168. Revised the 'Tag-to-retire Count' and 'Completion-to-retire count' on page 314. Clarified the use of the IbsDcStBnkCon and IbsDcLdBnkCon fields in IbsOpData3 register, clarified the description concerning data cache miss latency in 'Interpreting IBS Load/Store Data' on page 316.
December 2007	3.05	Added Appendix E, 'NUMA Optimizations for I/O Devices' on page 293. Added Appendix F, 'Remarks on the RDTSC(P) Instruction' on page 305
September 2007	3.04	Updated 'Memory and String Routines' on page 92; updated 'Code Padding with Operand-Size Override and Multibyte NOP' on page 68; updated Appendix C, 'Instruction Latencies' on page 245; Added Appendix G, 'Guide to Instruction-Based Sampling on AMD Family 10h and 12h Processors' on page 307.
June 2007	3.03	Made several minor corrections.
May 2007	3.02	Corrected several AMD-V™ related typos.
May 2007	3.01	Corrected the specified value for the MXCSR misaligned exception mask bit.

Date	Rev.	Description
April 2007	3.00	Initial release.

Chapter 1 Introduction

This guide provides optimization information and recommendations for AMD Family 10h and 12h processors. These optimizations are designed to yield software code that is fast, compact, and efficient. Toward this end, the optimizations in each of the following chapters are listed in order of importance.

This chapter covers the following topics:

Topic	Page
Intended Audience	1
Getting Started	1
Using This Guide	2
Important New Terms	4
Key Optimizations	6
What's New on AMD Family 10h and 12h Processors	6

1.1 Intended Audience

This book is intended for compiler and assembler designers, as well as C, C++, and assembly-language programmers writing performance-sensitive code sequences. This guide assumes that you are familiar with the AMD64 instruction set and the AMD64 architecture (registers and programming modes). For complete information on the AMD64 architecture and instruction set, see the multivolume *AMD64 Architecture Programmer's Manual* available from AMD.com. Individual volumes and their order numbers are provided below.

Title	Order Number
Volume 1, *Application Programming*	24592
Volume 2, *System Programming*	24593
Volume 3, *General-Purpose and System Instructions*	24594
Volume 4, *128-Bit Media Instructions*	26568
Volume 5, *64-Bit Media and x87 Floating-Point Instructions*	26569
AMD64 Architecture Programmer's Manual Documentation Updates	33633

1.2 Getting Started

More experienced readers may skip to "Key Optimizations" on page 6, which identifies the most important optimizations, and to "What's New on AMD Family 10h and 12h Processors" on page 6 for a quick review of key new performance enhancement features introduced with AMD Family 10h and 12h processors.

1.3 Using This Guide

Each of the remaining chapters in this document focuses on a particular general area of relevance to software optimization on AMD Family 10h and 12h processors. Each chapter is organized into a set of one or more recommended related optimizations pertaining to a particular issue. These sections are divided into three sections:

- Optimization—Specifies the recommended action required for achieving the optimization under consideration.
- Application—Specifies the type of software for which the particular optimization is relevant (*i.e.*, to 32-bit software or 64-bit software or to both).
- Rationale—Provides additional explanatory technical information regarding the particular optimization. This section usually provides illustrative C, C++, or assembly code examples as well.

The chapters that follow cover the following topics:

- Chapter 2, “C and C++ Source-Level Optimizations,” describes techniques that you can use to optimize your C and C++ source code.
- Chapter 3, “General 64-Bit Optimizations,” presents general assembly-language optimizations that can improve the performance of software designed to run in 64-bit mode. The optimizations in this chapter apply *only* to 64-bit software.
- Chapter 4, “Instruction-Decoding Optimizations,” discusses optimizations designed to maximize the number of instructions that the processor can decode at one time.
- Chapter 5, “Cache and Memory Optimizations,” discusses how to take advantage of the large L1 caches and high-bandwidth buses.
- Chapter 6, “Branch Optimizations,” discusses improving branch prediction and minimizing branch penalties.
- Chapter 7, “Scheduling Optimizations.” discusses improving instruction scheduling in the processor.
- Chapter 8, “Integer Optimizations,” discusses integer performance.
- Chapter 9, “Optimizing with SIMD Instructions,” discusses the 64-bit and 128-bit SIMD instructions (SSE, SSE2, SSE3, SSE4a) used to encode floating-point and integer operations.
- Chapter 10, “x87 Floating-Point Optimizations,” discusses optimizations using the x87 assembly instructions.
- Chapter 11, “Multiprocessor Considerations,” discusses processor/core selection and related issues for applications running on multiprocessor/multicore cache coherent non-uniform memory access (ccNUMA) configurations.
- Chapter 12, “Optimizing Secure Systems,” discusses ways to minimize the performance overhead imposed by the virtualization of a guest.

- Appendix A, “Microarchitecture of AMD Family 10h and 12h Processors,” discusses the internal design, or microarchitecture, of the AMD Family 10h and 12h processors and provides information about translation-lookaside buffers and other functional units that, while not part of the main processor, are integrated on the chip.
- Appendix B, “Implementation of Write-Combining,” describes how AMD Family 10h and 12h processors perform memory write-combining.
- Appendix C, “Instruction Latencies,” provides a complete listing of all AMD64 instructions with each instruction’s decode type, execution latency, and—where applicable—the pipes and throughput used in the floating-point unit.
- Appendix D, “Tools and APIs for AMD Family 10h and 12h ccNUMA Multiprocessor Systems” provides information on tools for programming in NUMA environments.
- Appendix E, “NUMA Optimizations for I/O Devices” provides information on the association of particular I/O devices with a specific nodes in a NUMA system.
- Appendix F, “Remarks on the RDTSC(P) Instruction” provides information on using the RDTSC and RDTSCP instructions to load the value of the time stamp counter (TSC).
- Appendix G, “Guide to Instruction-Based Sampling on AMD Family 10h and 12h Processors” describes the use of intruction-based sampling to monitor software performance.

1.3.1 Special Information

Special information in this guide is marked as follows:

❖ This symbol appears next to the most important, or key, optimizations.

1.3.2 Numbering Systems

The following suffixes identify different numbering systems:

This suffix	Identifies a
b	Binary number. For example, the binary equivalent of the number 5 is written 101b.
d	Decimal number. Decimal numbers are followed by this suffix only when the possibility of confusion exists. In general, decimal numbers are shown without a suffix.
h	Hexadecimal number. For example, the hexadecimal equivalent of the number 60 is written 3Ch.

1.3.3 Typographic Notation

This guide uses the following typographic notations for certain types of information:

This type of text	Identifies
italic	Placeholders that represent information you must provide. Italicized text is also used for the titles of publications and for emphasis.
`monowidth`	Program statements and function names.

1.4 Important New Terms

This section defines several important terms and concepts used in this guide.

1.4.1 Multi-Core Processors

The AMD Family 10h and 12h family of processors have multiple cores. A multi-core processor contains two to four identical cores that share the processor's cache and Northbridge (see Appendix A). Within a processor, each core can simultaneously run independent threads.

1.4.2 Primitive Operations

AMD Family 10h and 12h processors perform four types of *primitive operations*:

- Integer (arithmetic or logic)
- Floating-point (arithmetic)
- Load
- Store

1.4.3 Internal Instruction Formats

The AMD64 instruction set is complex. Instructions have variable-length encoding and many perform multiple primitive operations. AMD Family 10h and 12h processors do not execute these complex instructions directly, but, instead, decode them internally into simpler fixed-length instructions called *macro-ops*. Processor schedulers subsequently break down macro-ops into sequences of even simpler instructions called *micro-ops*, each of which specifies a single primitive operation.

A *macro-op* is a fixed-length instruction that:

- Expresses, at most, one integer or floating-point operation and one load and/or store operation.
- Is the primary unit of work managed (that is, dispatched and retired) by the processor.

A *micro-op* is a fixed-length instruction that:

- Expresses one and only one of the primitive operations that the processor can perform (for example, a load).
- Is executed by the processor's execution units.

Table 1 on page 5 summarizes the differences between AMD64 instructions, macro-ops, and micro-ops.

Table 1. Instructions, Macro-ops and Micro-ops

Comparing	AMD64 instructions	Macro-ops	Micro-ops
Complexity	Complex A single instruction may specify one or more of each of the following operations: • Integer or floating-point • Load • Store	Average A single macro-op may specify—at most—one integer or floating-point operation and one of the following operations: • Load • Store • Load and store to the same address	Simple A single micro-op specifies only one of the following primitive operations: • Integer or floating-point • Load • Store
Encoded length	Variable (instructions are different lengths)	Fixed (all macro-ops are the same length)	Fixed (all micro-ops are the same length)
Regularized instruction fields	No (field locations and definitions vary among instructions)	Yes (field locations and definitions are the same for all macro-ops)	Yes (field locations and definitions are the same for all micro-ops)

1.4.4 Types of Instructions

Instructions are classified according to how they are decoded by the processor. There are three types of instructions:

Instruction Type	Description
DirectPath Single	Decodes directly into one macro-op in microprocessor hardware.
DirectPath Double	Decodes directly into two macro-ops in microprocessor hardware.
VectorPath	Decodes into one or more (usually three or more) macro-ops using the on-chip microcode-engine ROM (MROM).

1.5 Key Optimizations

While all of the optimizations in this guide help improve software performance, some of them have more impact than others. Optimizations that offer the most improvement are called *key* optimizations.

❖ This symbol appears next to the most important (key) optimizations.

1.5.1 Implementation Guideline

Concentrate your efforts on implementing key optimizations before moving on to other optimizations.

Table 2 lists the key optimizations. These optimizations are discussed in detail in later sections of this book.

Table 2. Optimizations by Rank

Rank	Optimization
1	Load-Execute Instructions for Floating-Point or Integer Operands (See section 4.2 on page 53.)
2	Write-Combining (See section 5.6 on page 89.)
3	Branches That Depend on Random Data (See section 6.3 on page 101.)
4	Loop Unrolling (See section 7.2 on page 110.)
5	Pointer Arithmetic in Loops (See section 7.6 on page 116.)
6	Explicit Load Instructions (See section 9.2 on page 148.)
7	Reuse of Dead Registers (See section 9.15 on page 167.)
8	ccNUMA Optimizations (See section 11.1 on page 183.)
9	Multithreading (See section 11.3 on page 194.)
10	Prefetch and Streaming Instructions (See section 5.5 on page 81.)
11	Memory and String Routines (See section 5.9 on page 92.)
12	Loop Iteration Boundaries (See section 4.3 on page 56.)
13	Floating-Point Scalar Conversions (See sections 9.16 on page 168.)

1.6 What's New on AMD Family 10h and 12h Processors

AMD Family 10h and 12h processors introduce several new features that can significantly enhance software performance when compared to the previous AMD64 microprocessors. The following section provides compiler/assembler designers and C/C++/assembly language programmers with a summary of these performance improvements. Throughout this discussion, it is assumed that readers are familiar with the software optimization guide for the previous AMD64 processors and the terminology used there.

1.6.1 AMD Instruction Set Enhancements

The AMD Family 10h and 12h processors have been enhanced with the following new instructions:

- LZCNT, POPCNT—Advanced Bit Manipulation (ABM) instructions operate on general purpose registers.
- MOVNTSS, MOVNTSD, EXTRQ, INSERTQ—SSE4a instructions operate on XMM registers.

Support for these instructions is implementation dependent. See the *CPUID Specification*, order# 25481, and the *AMD64 Architecture Programmer's Manual Updates Application Note*, order# 33633, for additional information.

1.6.2 Floating-Point Improvements

Previous AMD64 processors supported 64-bit floating-point execution units. The new AMD Family 10h and 12h processors add support for 128-bit floating-point execution units. As a result, the throughput of both single-precision and double-precision floating-point SSE*x* vector operations has improved by 2X over the previous generation of AMD processors.

Performance Guidelines for Vectorized Floating-Point SSE*x* Code

While 128-bit floating-point execution units imply better performance for vectorized floating-point SSE*x* code, it is necessary to adhere to several performance guidelines to realize their full potential:

- Avoid writing less than 128 bits of an XMM register when using certain initializing and non-initializing operations.

 A floating-point XMM register is viewed as one 128-bit register internally by the processor. Writing to a 64-bit half of a 128-bit XMM register results in a merge dependency on the other 64-bit half. Therefore the following replacements are advised on AMD Family 10h processors:

 – Replace `MOVLPX/MOVHPX reg, mem` pairs with `MOVUPX reg, mem`, irrespective of the alignment of the data. On AMD Family 10h and 12h processors, the MOVUPX instruction is just as efficient as MOVAPX for when data is aligned. Hence it is advised to use MOVUPX regardless of the alignment.

 – Replace `MOVLPD reg, mem` with `MOVSD reg, mem`.

 – Replace `MOVSD reg, reg` with `MOVAPD reg, reg`.

 However, there are also several instructions that initialize the lower 64 or 32 bits of an XMM register and zero out the upper 64 or 96 bits and, thus, do not suffer from such merge dependencies. Consider, for example, the following instructions:

```
MOVSD xmm, [mem64]
MOVSS xmm, [mem32]
```

 When writing to a register during the course of a non-initializing operation on the register, there is usually no additional performance loss due to partial register reads and writes. This is because in

the typical case, the partial register that is being written is also a source to the operation. For example, `addsd xmm1, xmm2` does not suffer from merge dependencies.

There are often cases of non-initializing operations on a register, in which the partial register being written by the operation is not a source for the operation. In these cases also, it is preferable to avoid partial register writes. If it is not possible to avoid writing to a part of that register, then you should schedule any prior operation on any part of that register well ahead of the point where the partial write occurs.

Examples of non-initializing instructions that result in merge dependencies are SQRTSD, CVTPI2PS, CVTSI2SD, CVTSS2SD, MOVLHPS, MOVHLPS, UNPCKLPD and PUNPCKLQDQ.

For additional details on this optimization see “Partial-Register Writes” on page 60, “Explicit Load Instructions” on page 148, “Unaligned and Aligned Data Access” on page 149, and “Reuse of Dead Registers” on page 167.

- In the event of a load following a previous store to a given address for aligned floating-point vector data, use 128-bit stores and 128-bit loads instead of MOVLPX/MOVHPX pairs for storing and loading the data. This allows store-to-load forwarding to occur. Using MOVLPX/MOVHPX pairs is still recommended for storing unaligned floating-point vector data. Additional details on these restrictions can be obtained in “Store-to-Load Forwarding Restrictions” on page 74.
- To make use of the doubled throughput of both single-precision and double-precision floating-point SSEx vector operations, a compiler or an application developer can consider either increasing the unrolling factor of loops that include such vector operations and/or performing other code transformations to keep the floating-point pipeline fully utilized.

1.6.3 Load-Execute Instructions for Unaligned Data

Use load-execute instructions instead of discrete load and execute instructions when performing SSE floating-point/SSE integer/x87 computations on floating-point source operands. This is recommended regardless of the alignment of packed data on AMD Family 10h and 12h processors. (The use of load-execute instructions under these circumstances was only recommended for aligned packed data on the previous AMD64 processors.) This replacement is only possible if the misaligned exception mask (MM) is set. See the *AMD CPUID Specification*, order# 25481, and the *AMD64 Architecture Programmer’s Manual Updates Application Note*, order# 33633,for additional information on SSE misaligned access support. This optimization can be especially useful in vectorized SSE*x* loops and may eliminate the need for loop peeling due to nonalignment. (See “Load-Execute Instructions for Floating-Point or Integer Operands” on page 53.)

1.6.4 Instruction Fetching Improvements

The fetch window has changed from 16 bytes on previous AMD64 processors to 32 bytes on AMD Family 10h and 12h processors. The 32-byte fetch, when combined with the 128-bit floating-point execution unit, allows the processor to sustain a fetch/dispatch/retire sequence of three large instructions per cycle.

Assembly language programmers can now group large instructions together without worrying about an instruction possibly spanning the fetch window. In this regard, it is also advisable to align hot loops to 32 bytes instead of 16 bytes, especially in the case of loops for large SSE instructions.

For additional details, readers can refer to “Loop Iteration Boundaries” on page 56.

1.6.5 Instruction Decode and Floating-Point Pipe Improvements

Several integer and floating-point instructions have improved latencies and decode types on AMD Family 10h and 12h processors. Furthermore, the FPU pipes utilized by several floating-point instructions have changed. These changes can influence instruction choice and scheduling for compilers and hand-written assembly code. A comprehensive listing of all AMD64 instructions with their decode types, decode type changes from previous families of AMD processors, and execution latencies and FPU pipe utilization data are available in Appendix C.

1.6.6 Notable Performance Improvements

Several enhancements to the AMD64 architecture have resulted in significant performance improvements in AMD Family 10h and 12h processors, including:

- Improved performance of shuffle instructions
- Improved data transfer between floating-point registers and general purpose registers
- Improved floating-point register to floating-point register moves
- Optimization of repeated move instructions
- More efficient PUSH/POP stack operations
- 1-Gbyte paging

These are discussed in the following paragraphs and elsewhere in this document.

Improved Bandwidth Decode Type for Shuffle Instructions

The floating-point logic in AMD Family 10h and 12h processors uses three separate execution positions or pipes called FADD, FMUL and FSTORE. This is illustrated in Figure 8 on page 226 in Appendix A. Current AMD Family 10h and 12h processors support two SSE logical/shuffle units, one in the FMUL pipe and another in the FADD pipe, while previous AMD64 processors have only one SSE logical/shuffle unit in the FMUL pipe. As a result, the SSE/SSE2 shuffle instructions can be processed at twice the previous bandwidth on AMD Family 10h and 12h processors. Furthermore, the PSHUFD and SHUFP*x* shuffle instructions are now DirectPath instructions instead of VectorPath instructions on AMD Family 10h and 12h processors and take advantage of the 128-bit floating point execution units. Hence, these instructions get a further 2X boost in bandwidth, resulting in an overall improvement of 4X in bandwidth compared to the previous generation of AMD processors.

It’s more efficient to use SHUFP*x* and PSHUFD instructions over combinations of more than one MOVLHPS/MOVHLPS/UNPCK*x*/PUNPCK*x* instructions to do shuffle operations.

Data Transfer Between Floating-Point Registers and General Purpose Integer Registers

We recommend using the MOVD instruction when moving data from an MMX™ or XMM register to a GPR. However, when moving data from a GPR to an MMX or XMM floating-point register, it is advisable to use separate store and load instructions to move the data from the source register to a temporary location in memory and then from memory into the destination register, taking the memory latency into account when scheduling them.

The performance of the CVTSS2SI, CVTTSS2SI, CVTSD2SI, CVTTSD2SI instructions that are used to convert floating-point data to integer data has improved. For additional details see “Floating-Point-to-Integer Conversion” on page 167.

Floating-Point Register-to-Register Moves

On previous AMD processors, floating-point register-to-register moves could only go through the FADD and FMUL pipes. On AMD Family 10h and 12h processors, floating-point register-to-register moves can also go through the FSTORE pipe, thereby improving overall throughput.

Repeated String Instructions

REP instructions have been optimized on AMD Family 10h and 12h processors. See “Repeated String Instructions” on page 128 for details on how to take advantage of these optimizations.

Faster PUSH/POP with the Sideband Stack Optimizer

AMD Family 10h and 12h processors have added a sideband stack optimizer (SSO). This special circuitry removes the dependency that arises during chains of PUSH and POP operations on the rSP register and, thereby, improves the efficiency of the PUSH and POP instructions.

The SSO also improves the performance of CALL and RET instructions, among others. (See “Stack Operations” on page 59.)

1.6.7 Large Page Support

AMD Family 10h and 12h processors now have better large page support, having incorporated new 1GB paging and 2MB and 4KB paging improvements.

The L1 data TLB and L2 data TLB now support 1GB pages, a benefit to applications making large data-set random accesses.

The L1 instruction TLB, L1 data TLB and L2 data TLB have increased the number of entries for 2MB pages. This improves the performance of software that uses 2MB code or data or code mixed with data virtual pages.

The L1 data TLB has also increased the number of entries for 4KB pages.

For additional details on the actual number of TLB entries, see section A.10, “Translation-Lookaside Buffer” on page 230.

For more information on 1-Gbyte pages see “Using 1-Gbyte Virtual Memory Pages” on page 96.

1.6.8 AMD Virtualization™ Optimizations

Chapter 12, “Optimizing Secure Virtual Machines” covers optimizations that minimize the performance overhead imposed by the virtualization of a guest in AMD Virtualization™ technology (AMD-V™). Topics include:

- The advantages of using nested paging instead of shadow paging
- Guest page attribute table (PAT) configuration
- State swapping
- Economizing Interceptions
- Nested page and shadow page size
- TLB control and flushing in shadow pages
- Instruction Fetch for Intercepted (REP) INS instructions
- Sharing IOIO and MSR protection masks
- CPUID
- Time resources
- Paravirtualized resources

Chapter 2 C and C++ Source-Level Optimizations

Although C and C++ compilers can often produce very efficient object code from naive source code, careful attention to coding details can lead to even better object code and therefore to improved performance. Many optimizations take advantage of the underlying mechanisms used by C and C++ compilers to translate source code into sequences of AMD64 instructions. This chapter includes guidelines for writing C and C++ source code that yields an approximation to the most highly efficient optimization.

This chapter covers the following topics:

Topic	Page
Declarations of Floating-Point Values	14
Using Arrays and Pointers	14
Unrolling Small Loops	17
Arrange Boolean Operands for Quick Expression Evaluation	18
Expression Order in Compound Branch Conditions	19
Long Logical Expressions in If Statements	20
Dynamic Memory Allocation Consideration	21
Unnecessary Store-to-Load Dependencies	21
Matching Store and Load Size	23
Use of Function Prototypes	25
Use of const Type Qualifier	26
Generic Loop Hoisting	26
Local Static Functions	28
Explicit Parallelism in Code	29
Extracting Common Subexpressions	30
Sorting and Padding C and C++ Structures	31
Replacing Integer Division with Multiplication	32
Frequently Dereferenced Pointer Arguments	33
32-Bit Integral Data Types	34
Sign of Integer Operands	35
Accelerating Floating-Point Division and Square Root	36
Speeding Up Branches Based on Comparisons Between Floats	38
Improving Performance in Linux® Libraries	40
Aligning Matrices	41

2.1 Declarations of Floating-Point Values

Optimization

When working with single precision (`float`) values:

- Use the `f` or `F` suffix (for example, `3.14f`) to specify a constant value of type `float`.
- Use function prototypes for all functions that accept arguments of type `float`.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

C and C++ compilers treat floating-point constants and arguments as double precision (`double`) unless you specify otherwise. However, single precision floating-point values occupy half the memory space as double precision values and can often provide the precision necessary for a given computational problem.

This optimization also results in more efficient use of the XMM Streaming SIMD registers: four single precision values can be packed into a single XMM register, compared to two double precision values.

2.2 Using Arrays and Pointers

Optimization

Use array notation instead of pointer notation when working with arrays.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

C allows the use of either the array operator (`[]`) or pointers to access the elements of an array. However, the use of pointers in C makes work difficult for optimizers in C compilers. Without

detailed and aggressive pointer analysis, the compiler has to assume that writes through a pointer can write to any location in memory, including storage allocated to other variables. (For example, `*p` and `*q` can refer to the same memory location, while `x[0]` and `x[2]` cannot.) Pointers make it difficult for compilers to detect the presence or absence of aliasing—with possible ambiguous access to a block of memory. The compiler sometimes must assume aliasing in the presence of pointers, which limits the opportunities for optimization. Array notation makes the task of the optimizer easier by reducing possible aliasing.

Example

Avoid code, such as the following, which uses pointer notation:

```
typedef struct {
   float x, y, z, w;
} VERTEX;

typedef struct {
   float m[4][4];
} MATRIX;

void XForm(float *res, const float *v, const float *m, int numverts) {

   float dp;
   int i;
   const VERTEX* vv = (VERTEX *)v;

   for (i = 0; i < numverts; i++) {
      dp  = vv->x * *m++;
      dp += vv->y * *m++;
      dp += vv->z * *m++;
      dp += vv->w * *m++;

      *res++ = dp;  // Write transformed x.

      dp  = vv->x * *m++;
      dp += vv->y * *m++;
      dp += vv->z * *m++;
      dp += vv->w * *m++;

      *res++ = dp;  // Write transformed y.

      dp  = vv->x * *m++;
      dp += vv->y * *m++;
      dp += vv->z * *m++;
      dp += vv->w * *m++;

      *res++ = dp;  // Write transformed z.

      dp  = vv->x * *m++;
      dp += vv->y * *m++;
      dp += vv->z * *m++;
      dp += vv->w * *m++;
```

```
      *res++ = dp;  // Write transformed w.

      ++vv;     // Next input vertex
      m -= 16;  // Reset to start of transform matrix.
   }
}
```

Instead, use the equivalent array notation:

```
typedef struct {
    float x, y, z, w;
} VERTEX;

typedef struct {
    float m[4][4];
} MATRIX;

void XForm(float *res, const float *v, const float *m, int numverts) {
    int i;
    const VERTEX* vv = (VERTEX *)v;
    const MATRIX* mm = (MATRIX *)m;
    VERTEX* rr = (VERTEX *)res;
    for (i = 0; i < numverts; i++) {
        rr[i].x = vv[i].x * mm->m[0][0] + vv[i].y * mm->m[0][1] +
            vv[i].z * mm->m[0][2] + vv[i].w * mm->m[0][3];
        rr[i].y = vv[i].x * mm->m[1][0] + vv[i].y * mm->m[1][1] +
            vv[i].z * mm->m[1][2] + vv[i].w * mm->m[1][3];
        rr[i].z = vv[i].x * mm->m[2][0] + vv[i].y * mm->m[2][1] +
            vv[i].z * mm->m[2][2] + vv[i].w * mm->m[2][3];
        rr[i].w = vv[i].x * mm->m[3][0] + vv[i].y * mm->m[3][1] +
            vv[i].z * mm->m[3][2] + vv[i].w * mm->m[3][3];
    }
}
```

Additional Considerations

Source-code transformations interact with a compiler's code generator, making it difficult to control the generated machine code from the source level. It is even possible that source-code transformations aimed at improving performance may conflict with compiler optimizations. Depending on the compiler and the specific source code, it is possible for pointer-style code to compile into machine code that is faster than that generated from equivalent array-style code. Compare the performance of your code after implementing a source-code transformation with the performance of the original code to be sure that there is an improvement.

Some compilers provide proprietary declaration keywords that further allow the compiler to reduce possible aliasing. See *Compiler Usage Guidelines for 64-Bit Operating Systems on AMD64 Platforms Application Note*, order# 32035, for details.

2.3 Unrolling Small Loops

Optimization

Completely unroll loops that have a small fixed loop count and a small loop body.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Many compilers do not aggressively unroll loops. Manually unrolling loops can benefit performance, especially if the loop body is small, making the loop overhead significant.

Unrolling loops increases the code size, which may decrease performance in rare cases.

Example

Avoid a small loop like this:

```
// 3D-transform: Multiply vector V by 4x4 transform matrix M.
for (i = 0; i < 4; i++) {
   r[i] = 0;
   for (j = 0; j < 4; j++) {
      r[i] += m[j][i] * v[j];
   }
}
```

Instead, replace it with its completely unrolled equivalent, as shown here:

```
r[0] = m[0][0] * v[0] + m[1][0] * v[1] + m[2][0] * v[2] + m[3][0] * v[3];
r[1] = m[0][1] * v[0] + m[1][1] * v[1] + m[2][1] * v[2] + m[3][1] * v[3];
r[2] = m[0][2] * v[0] + m[1][2] * v[1] + m[2][2] * v[2] + m[3][2] * v[3];
r[3] = m[0][3] * v[0] + m[1][3] * v[1] + m[2][3] * v[2] + m[3][3] * v[3];
```

Related Information

For information on loop unrolling at the assembly-language level, see “Loop Unrolling” on page 110.

2.4 Arrange Boolean Operands for Quick Expression Evaluation

Optimization

In expressions that use the logical AND (&&) or logical OR (||) operator, arrange the operands for quick evaluation of the expression:

If the expression uses this operator	Then arrange the operands from left to right in decreasing probability of being
&& (logical AND)	False
\|\| (logical OR)	True

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

C and C++ compilers guarantee short-circuit evaluation of the boolean operators && and ||. In an expression that uses &&, the first operand to evaluate to false terminates the evaluation; subsequent operands are not evaluated. In an expression that uses ||, the first operand to evaluate to true terminates the evaluation.

When used to control program flow, expressions involving && and || are translated into a series of conditional branches. This optimization minimizes the total number of conditions evaluated and branches executed.

Example 1

In the following code, the operands of && are not arranged for quick expression evaluation because the first operand is not the condition case most likely to be false (it is far less likely for an animal name to begin with a ‘y’ than for it to have fewer than four characters):

```
char animalname[30];
char *p;

p = animalname;

if ((strlen(p) > 4) && (*p == 'y')) { ... }
```

Because the odds that the animal name begins with a ‘y’ are comparatively low, it is better to put that operand first:

```
if ((*p == 'y') && (strlen(p) > 4)) { ... }
```

Example 2

In the following code (assuming a uniform random distribution of `i`), the operands of `||` are not arranged for quick expression evaluation because the first operand is not the condition most likely to be true:

```
unsigned int i;

if ((i < 4) || (i & 1)) { ... }
```

Because it is more likely for the least-significant bit of `i` to be 1, it is better to put that operand first:

```
if ((i & 1) || (i < 4)) { ... }
```

2.5 Expression Order in Compound Branch Conditions

Optimization

In the most active areas of a program, order the expressions in compound branch conditions to take advantage of short circuiting of compound conditional expressions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Branch conditions in C programs often consist of compound conditions consisting of multiple boolean expressions joined by the logical AND (`&&`) or logical OR (`||`) operators. C compilers guarantee short-circuit evaluation of these operators. In a compound logical OR expression, the first operand to evaluate to true terminates the evaluation, and subsequent operands are not evaluated at all. Similarly, in a logical AND expression, the first operand to evaluate to false terminates the evaluation. Hence, it is not always possible to swap the operands of logical OR and logical AND. This is especially true when the evaluation of one of the operands causes a side effect. In most cases the order of operands in such expressions is irrelevant.

When used to control conditional branches, expressions involving logical OR or logical AND are translated into a series of conditional branches. The ordering of the conditional branches is a function of the ordering of the expressions in the compound condition and can have a significant impact on

performance. It is impossible to give an easy, closed-form formula on how to order the conditions. Overall performance is a function of the following factors:

- Probability of a branch misprediction for each of the branches generated
- Additional latency incurred due to a branch misprediction
- Cost of evaluating the conditions controlling each of the branches generated
- Amount of parallelism that can be extracted in evaluating the branch conditions
- Data stream consumed by an application (mostly due to the dependence of misprediction probabilities on the nature of the incoming data in data-dependent branches)

It is recommended to experiment with the ordering of expressions in compound branch conditions in the most active areas of a program (“hot spots,” consuming a great amount of execution time). Such hot spots can be found through the use of profiling by feeding a typical data stream to the program while doing the experiments.

2.6 Long Logical Expressions in If Statements

Optimization

In `if` statements, avoid long logical expressions that can generate dense conditional branches that violate the guideline described in “Branch Alignment and Density” on page 99. When long logical expressions are unavoidable, try to arrange them so that most of the implicit branches are not be taken.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

High branch density can lead to some branches not being identified by the branch predictor (as described in section “Branch Alignment and Density” on page 99). If these unpredicted branches are not taken, they will not cause a misprediction penalty.

Preferred for Data that Falls Mostly Within the Range

```
if (a <= max && a >= min && b <= max && b >= min)
```

If most of the data falls within the range, the branches will not be taken, so the above code is preferred. Otherwise, the following code is preferred.

Preferred for Data that Does Not Fall Mostly Within the Range

```
if (a > max || a < min || b > max || b < min)
```

2.7 Dynamic Memory Allocation Consideration

Optimization

Where this aligned pointer cannot be guaranteed, use the technique shown in the following code to make the pointer 16-byte aligned, if needed.

Application

This optimization applies to:

- 32-bit software

Examples

Dynamic memory allocation—accomplished through the use of the `malloc` library function in C—should always return a pointer that is suitably aligned for the largest base type (16-byte alignment). However, this may not always be the case. In this example, after memory allocation, use `np` instead of `p` to access the data. The pointer `p` is still needed in order to deallocate the storage later.

```
double *p;
double *np;

p = (double *)malloc(sizeof(double) * number_of_doubles + 15);
np = (double *)((((ptrdiff_t)(p)) + 15L) & (-16L));
```

2.8 Unnecessary Store-to-Load Dependencies

Optimization

Avoid store-to-load dependencies.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

A store-to-load dependency exists when data is stored to memory, only to be read back shortly thereafter. For details, see “Store-to-Load Forwarding Restrictions” on page 74. The

AMD Family 10h and 12h processors contain hardware to accelerate such store-to-load dependencies, allowing the load to obtain the store data before it has been written to memory. However, avoiding such dependencies and keeping the data in an internal register results in faster code.

It is especially important to avoid store-to-load dependencies if they are part of a long dependency chain, as may occur in a recurrence computation. If the dependency occurs while operating on arrays, many compilers are unable to optimize the code in a way that avoids the store-to-load dependency. In some instances the language definition may prohibit the compiler from using code transformations that would remove the store-to-load dependency. Therefore, it is recommended that the programmer remove the dependency manually, for example, by introducing a temporary variable that can be kept in a register. This can result in a significant performance increase.

Examples

Avoid

```
double x[VECLEN], y[VECLEN], z[VECLEN];
unsigned int k;

for (k = 1; k < VECLEN; k++) {
   x[k] = x[k-1] + y[k];
}

for (k = 1; k < VECLEN; k++) {
   x[k] = z[k] * (y[k] - x[k-1]);
}
```

Preferred

```
double x[VECLEN], y[VECLEN], z[VECLEN];
unsigned int k;
double t;

t = x[0];
for (k = 1; k < VECLEN; k++) {
   t = t + y[k];
   x[k] = t;
}

t = x[0];
for (k = 1; k < VECLEN; k++) {
   t = z[k] * (y[k] - t);
   x[k] = t;
}
```

2.9 Matching Store and Load Size

Optimization

Align memory accesses and match addresses and sizes of stores and dependent loads.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The AMD Family 10h and 12h processors contain a load-store buffer to speed up the forwarding of store data to dependent loads. However, this store-to-load forwarding (STLF) inside the load-store buffer occurs, in general, only when the addresses and sizes of the store and the dependent load match, and when both memory accesses are aligned. For details, see “Store-to-Load Forwarding Restrictions” on page 74.

It is impossible to control load and store activity at the source level in such a way as to avoid all cases that violate restrictions placed on store-to-load-forwarding. In some instances it is possible to spot such cases in the source code. Size mismatches can easily occur when different-size data items are joined in a union. Address mismatches could be the result of pointer manipulation.

The following examples show a situation involving a union of different-size data items. The examples show a user-defined unsigned 16.16 fixed-point type and two operations defined on this type. Function `fixed_add` adds two fixed-point numbers, and function `fixed_int` extracts the integer portion of a fixed-point number. Listing shows an inappropriate implementation of `fixed_int`, which, when used on the result of `fixed_add`, causes misalignment, address mismatch, or size mismatch between memory operands, such that no store-to-load forwarding in the load-store buffer takes place. The following examples shows how to properly implement `fixed_int` in order to allow store-to-load forwarding in the load-store buffer.

Examples

Avoid

```
typedef union {
   unsigned int whole;
   struct {
      unsigned short frac; /* Lower 16 bits are fraction. */
      unsigned short intg; /* Upper 16 bits are integer.  */
   } parts;
} FIXED_U_16_16;
```

```
__inline FIXED_U_16_16 fixed_add(FIXED_U_16_16 x, FIXED_U_16_16 y) {
     FIXED_U_16_16 z;
     z.whole = x.whole + y.whole;
     return (z);
}

__inline unsigned int fixed_int(FIXED_U_16_16 x) {
     return((unsigned int)(x.parts.intg));
}
...
FIXED_U_16_16 y, z;
unsigned int q;
...
label1:
y = fixed_add (y, z);
q = fixed_int (y);

label2:
...
```

The object code generated for the source code between `label1` and `label2` typically follows one of these two variants:

```
; Variant 1
mov edx, DWORD PTR [z]
mov eax, DWORD PTR [y]      ; -+
add eax, edx                ;  |
mov DWORD PTR [y], eax      ;  |
mov EAX, DWORD PTR [y+2]    ; <+ Address mismatch--no forwarding in LSU
and EAX, 0FFFFh
mov DWORD PTR [q], eax

; Variant 2
mov   edx, DWORD PTR [z]
mov   eax, DWORD PTR [y]     ; -+
add   eax, edx               ;  |
mov   DWORD PTR [y], eax     ;  |
movzx eax, WORD PTR [y+2]    ; <+ Size and address mismatch--no forwarding in LSU
mov   DWORD PTR [q], eax
```

Some more sophisticated compilers may generate optimal machine code even for the previous example. These compilers provide various optional levels and types of optimizations that are controlled by compiler program flags. When compiled at a moderate level of optimization, such compilers may generate perfectly acceptable code from C++ code such as that listed above. For more information, see *Compiler Usage Guidelines for 64-Bit Operating Systems on AMD64 Platforms Application Note*, order# 32035.

Preferred

```
typedef union {
   unsigned int whole;
   struct {
      unsigned short frac; /* Lower 16 bits are fraction. */
      unsigned short intg; /* Upper 16 bits are integer.  */
```

```
   } parts;
} FIXED_U_16_16;

__inline FIXED_U_16_16 fixed_add(FIXED_U_16_16 x, FIXED_U_16_16 y) {
   FIXED_U_16_16 z;
   z.whole = x.whole + y.whole;
   return(z);
}

__inline unsigned int fixed_int(FIXED_U_16_16 x) {
   return (x.whole >> 16);
}
...
FIXED_U_16_16 y, z;
unsigned int q;
...
label1:
y = fixed_add (y, z);
q = fixed_int (y);

label2:
...
```

The object code generated for the source code between `label1` and `label2` typically looks like this:

```
mov edx, DWORD PTR [z]
mov eax, DWORD PTR [y]
add eax, edx
mov DWORD PTR [y], eax   ; -+
mov eax, DWORD PTR [y]   ; <+ Aligned (size/address match)--forwarding in LSU
shr eax, 16
mov DWORD PTR [q], eax
```

2.10 Use of Function Prototypes

Optimization

In general, use prototypes for all functions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Prototypes can convey additional information to the compiler that might enable more aggressive optimizations.

2.11 Use of const Type Qualifier

Optimization

For objects whose values will not be changed, use the `const` type qualifier.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Using the `const` type qualifier makes code more robust and may enable the compiler to generate higher-performance code. For example, under the C standard, a compiler is not required to allocate storage for an object that is declared `const`, if its address is never used.

2.12 Generic Loop Hoisting

Optimization

To improve the performance of inner loops, reduce redundant constant calculations (that is, loop-invariant calculations). This idea can also be extended to invariant control structures.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale and Examples

The following example demonstrates the use of an invariant condition in an `if` statement in a `for` loop. The second listing shows the preferred optimization.

Avoid

```
for (i...) {
   if (CONSTANT0) {
      DoWork0(i);   // Does not affect CONSTANT0.
   }
   else {
```

```
      DoWork1(i);   // Does not affect CONSTANT0.
   }
}
```

Preferred

```
if (CONSTANT0) {
   for (i...) {
      DoWork0(i);
   }
}
else {
   for (i...) {
      DoWork1(i);
   }
}
```

The preferred optimization in the preceding example tightens the inner loops by avoiding repetitious evaluation of a known `if` control structure. Although the branch would be easily predicted, the extra instructions and decode limitations imposed by branching are eliminated.

To generalize the preceding example further for multiple-constant control code, more work may be needed to create the proper outer loop. Enumeration of the constant cases reduces this to a simple `switch` statement.

Avoid

```
for (i...) {
   if (CONSTANT0) {
      DoWork0(i);   // Does not affect CONSTANT0 or CONSTANT1.
   }
   else {
      DoWork1(i);   // Does not affect CONSTANT0 or CONSTANT1.
   }

   if (CONSTANT1) {
      DoWork2(i);   // Does not affect CONSTANT0 or CONSTANT1.
   }
   else {
      DoWork3(i);   // Does not affect CONSTANT0 or CONSTANT1.
   }
}
```

Transform the loop in the preceding example (by using the `switch` statement) into:

Preferred

```
#define combine(c1, c2) (((c1) << 1) + (c2))
switch (combine(CONSTANT0 != 0, CONSTANT1 != 0)) {
   case combine(0, 0):
      for(i...) {
         DoWork0(i);
         DoWork2(i);
      }
```

```
      break;
   case combine(1, 0):
      for(i...) {
         DoWork1(i);
         DoWork2(i);
      }
      break;
   case combine(0, 1):
      for(i...) {
         DoWork0(i);
         DoWork3(i);
      }
      break;
   case combine( 1, 1 ):
      for(i...) {
         DoWork1(i);
         DoWork3(i);
      }
      break;
   default:
      break;
}
```

Some introductory code is necessary to generate all the combinations for the `switch` constant and the total amount of code has doubled. However, the inner loops are now free of `if` statements. In ideal cases where the `DoWorkn` functions are inlined, the successive functions have greater overlap, leading to greater parallelism than possible in the presence of intervening `if` statements.

The same idea can be applied to constant `switch` statements or to combinations of `switch` statements and `if` statements inside of `for` loops. The method used to combine the input constants becomes more complicated but benefits performance.

However, the number of inner loops can also substantially increase. If the number of inner loops is prohibitively high, then only the most common cases must be dealt with directly, and the remaining cases can fall back to the old code in the default clause of the `switch` statement. This situation is typical of run-time generated code. While the performance of run-time generated code can be improved by means similar to those presented here, it is much harder to maintain and developers must do their own code-generation optimizations without the help of an available compiler.

2.13 Local Static Functions

Optimization

Declare as `static` functions that are not used outside the file where they are defined.

Application

This optimization applies to:

- 32-bit software

- 64-bit software

Rationale

Declaring a function as `static` forces internal linkage. Functions that are not declared as `static` default to external linkage, which may inhibit certain optimizations—for example, aggressive inlining—with some compilers. In C++, programmers can declare functions inside an anonymous namespace to achieve the same local scoping effect.

2.14 Explicit Parallelism in Code

Optimization

Where possible, break long dependency chains into several independent chains that can be executed in parallel to take advantage of the execution units in each pipeline.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

It is especially important to break long x87, SSE, or SSE2 dependency chains into smaller executing units in floating-point code, because of the longer latency of floating-point operations. Most languages (including ANSI C) are bound by the guarantee that floating-point expressions can not be reordered; compilers cannot usually perform such optimizations unless they offer a switch to allow noncompliant reordering of floating-point expressions according to algebraic rules.

Reordered code that is algebraically identical to the original code does not necessarily produce identical computational results due to the lack of associativity of floating-point operations. There are well-known numerical considerations in applying these optimizations (consult a book on numerical analysis). In some cases, reordered floating-point code may lead to unexpected results, but in the vast majority of cases, the final result differs only in the least-significant bits.

Examples

Avoid

```
double a[100], sum;
int i;

sum = 0.0f;
```

```
for (i = 0; i < 100; i++) {
   sum += a[i];
}
```

Preferred

```
double a[100], sum1, sum2, sum3, sum4, sum;
int i;

sum1 = 0.0;
sum2 = 0.0;
sum3 = 0.0;
sum4 = 0.0;
for (i = 0; i < 100; i + 4) {
   sum1 += a[i];
   sum2 += a[i+1];
   sum3 += a[i+2];
   sum4 += a[i+3];
}
sum = (sum4 + sum3) + (sum1 + sum2);
```

Notice that the four-way unrolling is chosen to exploit the four-stage fully pipelined floating-point adder. Each stage of the floating-point adder is occupied on every clock cycle, ensuring maximum sustained utilization.

2.15 Extracting Common Subexpressions

Optimization

Manually extract common subexpressions from floating-point expressions, where C compilers may be unable to extract them due to the rules against reordering of floating-point expressions in the ANSI standard.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Specifically, the compiler cannot rearrange the computation according to algebraic equivalencies before extracting common subexpressions. Rearranging the expression may give different computational results due to the lack of associativity of floating-point operations, but the results usually differ in only the least-significant bits. However, since errors in the least significant bits can be magnified by later operations to the extent that they completely invalidate the calculation, the programmer should proceed with caution when implementing this sort of computation.

Examples

Avoid

```
double a, b, c, d, e, f;

e = b * c / d;
f = b / d * a;
```

Preferred

```
double a, b, c, d, e, f, t;

t = b / d;
e = c * t;
f = a * t;
```

Avoid

```
double a, b, c, e, f;

e = a / c;
f = b / c;
```

Preferred

```
double a, b, c, e, f, t;

t = 1 / c;
e = a * t
f = b * t;
```

2.16 Sorting and Padding C and C++ Structures

Optimization

Sort and pad C and C++ structures to achieve natural alignment.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

In order to achieve better alignment for structures, many compilers have options that allow padding of structures to make their sizes multiples of words, doublewords, or quadwords. In addition, to improve the alignment of structure members, some compilers may allocate structure elements in an order that differs from the order in which they are declared. Unfortunately, some compilers may not offer any of these features, or their implementations might not work properly in all situations.

By sorting and padding structures at the source-code level, if the first member of a structure is naturally aligned, then all other members are naturally aligned as well. This allows, for example, arrays of structures to be perfectly aligned.

Sorting and Padding C and C++ Structures

To sort and pad a C or C++ structure, follow these steps:

1. Sort the structure members according to their type sizes, declaring members with larger type sizes ahead of members with smaller type sizes.
2. Pad the structure so the size of the structure is a multiple of the largest member's type size.

Examples

Avoid structure declarations in which the members are not declared in order of their type sizes and the size of the structure is not a multiple of the size of the largest member's type:

```
struct {
   char a[5];    \\ Smallest type size (1 byte * 5)
   long k;       \\ 4 bytes in this example
   double x;     \\ Largest type size (8 bytes)
} baz;
```

Instead, declare the members according to their type sizes (largest to smallest) and add padding to ensure that the size of the structure is a multiple of the largest member's type size:

```
struct {
   double x;       \\ Largest type size (8 bytes)
   long k;         \\ 4 bytes in this example
   char a[5];      \\ Smallest type size (1 byte * 5)
   char pad[7];    \\ Make structure size a multiple of 8.
} baz;
```

2.17 Replacing Integer Division with Multiplication

Optimization

Replace integer division with multiplication when there are multiple divisions in an expression. (This is possible only if no overflow will occur during the computation of the product. The possibility of an overflow can be determined by considering the possible ranges of the divisors.)

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Integer division is the slowest of all integer arithmetic operations.

Examples

Avoid code that uses two integer divisions:

```
int i, j, k, m;

m = i / j / k;
```

Instead, replace one of the integer divisions with the appropriate multiplication:

```
m = i / (j * k);
```

2.18 Frequently Dereferenced Pointer Arguments

Optimization

Avoid dereferenced pointer arguments inside a function.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Because the compiler has no knowledge of aliasing between pointers, such dereferencing cannot be "optimized away." Since data may not be maintained in registers, memory traffic can significantly increase.

Many compilers have an "assume no aliasing" optimization switch. This allows the compiler to assume that two different pointers always have disjoint contents and does not require copying of pointer arguments to local variables. If your compiler does not have this type of optimization, then copy the data referenced by the pointer arguments to local variables at the start of the function and if necessary copy them back at the end of the function. (Some compilers also provide keywords to

provide the same aliasing information to the compiler. For details, see *Compiler Usage Guidelines for 64-Bit Operating Systems on AMD64 Platforms Application Note*, order# 32035.)

Examples

Avoid

```
// Assumes pointers are different and q != r.
void isqrt(unsigned long a, unsigned long *q, unsigned long *r) {

   *q = a;
   if (a > 0) {
      while (*q > (*r = a / *q)) {
         *q = (*q + *r) >> 1;
      }
   }
   *r = a - *q * *q;
}
```

Preferred

```
// Assumes pointers are different and q != r.
void isqrt(unsigned long a, unsigned long *q, unsigned long *r) {

   unsigned long qq, rr;
   qq = a;
   if (a > 0) {
      while (qq > (rr = a / qq)) {
         qq = (qq + rr) >> 1;
      }
   }
   rr = a - qq * qq;
   *q = qq;
   *r = rr;
}
```

2.19 32-Bit Integral Data Types

Optimization

Use 32-bit integers instead of smaller sized integers (16-bit or 8-bit).

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

When choosing between 32-bit, 16-bit and 8-bit data types in cases where memory footprint is not a concern, using 32-bit integer types in 32-bit software (32-bit or 64-bit integer types in 64-bit software) avoids possible register-merging false dependencies due to partial register writes. See section 4.8, "Partial-Register Writes" on page 60 for details.

2.20 Sign of Integer Operands

Optimization

Where there is a choice of using either a signed or an unsigned type, take into consideration that some operations are faster with unsigned types while others are faster for signed types.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

In many cases, the type of data to be stored in an integer variable determines whether a signed or an unsigned integer type is appropriate. For example, to record the weight of a person in pounds, no negative numbers are required, so an unsigned type is appropriate. However, recording temperatures in degrees Celsius may require both positive and negative numbers, so a signed type is needed.

Integer-to-floating-point conversion using integers larger than 16 bits is faster with signed types, as the AMD64 architecture provides instructions for converting signed integers to floating-point but has no instructions for converting unsigned integers. In a typical case, a 32-bit integer is converted by a compiler to assembly as follows:

Examples

Avoid

```
double x;          ====>   mov [temp+4], 0
unsigned int i;            mov eax, i
                           mov [temp], eax
x = i;                     fild QWORD PTR [temp]
                           fstp QWORD PTR [x]
```

The preceding code is slow not only because of the number of instructions, but also because a size mismatch prevents store-to-load forwarding to the FILD instruction. Instead, use the following code:

Preferred

```
double x;   ====>   fild DWORD PTR [i]
int i;              fstp QWORD PTR [x]

x = i;
```

Computing quotients and remainders in integer division by constants is faster when performed on unsigned types. The following typical case is the compiler output for a 32-bit integer divided by 4:

Avoid

```
int i;       ====>   mov eax, i
                     cdq
i = i / 4;           and edx, 3
                     add eax, edx
                     sar eax, 2
                     mov i, eax
```

Preferred

```
unsigned int i;   ====>   shr  i, 2

i = i / 4;
```

In summary, use unsigned types for:

- Division and remainders
- Loop counters
- Array indexing

Use signed types for:

- Integer-to-floating-point conversion

2.21 Accelerating Floating-Point Division and Square Root

Optimization

In applications employing the heavy use of single precision division and square root operations, in which the compiler maps floating-point operations to x87 instructions, the x87 FPU control word register precision control specification bits (PC) can be set to single precision to improve performance. (The processor defaults to double-extended precision. See *AMD64 Architecture Programmer's Manual, Volume 1,* order# 24592, for details on the FPU control register.)

Note: *For hotspots that can be recoded in assembly language or SSE intrinsics, refer to section 9.7"Using SIMD Instructions for Fast Square Roots and Divisions" on page 152 for coding suggestions.*

Application

This optimization applies to any compiler that maps floating-point operations to x87 instructions. This is generally true only for 32-bit compilers.

Rationale

Division and square root have a much longer latency than other floating-point operations, even though the AMD Family 10h and 12h processors provide significant acceleration of these two operations. In some application programs, these operations occur so often that they seriously impair performance. If the code has hot spots that use single precision arithmetic only (that is, all computation involves data of type `float`) and for some reason cannot be ported to SSE, SSE2, or SSE3 code, the following technique may be used to improve performance.

The x87 FPU has a precision-control field as part of the FPU control word. The precision-control setting determines rounding precision of instruction results and affects the basic arithmetic operations, including division and the extraction of square root. Division and square root on the AMD Family 10h and 12h processors are only computed to the number of bits necessary for the currently selected precision. Setting precision control to single precision (versus the default of double-extended precision) lowers the latency of those operations.

For example, the 32-bit version Microsoft® Visual C environment provides functions to manipulate the FPU control word and, thus, the precision control bits. Note that these functions are not very fast, so insert changes of precision control where doing so creates little overhead, such as outside of computation-intensive loops. Otherwise, the overhead created by the function calls outweighs the benefit of reducing the latencies of divide and square-root operations. The following example shows how to set the precision control to single precision and later restore the original settings in the Microsoft Visual C environment.

Examples

```
/* Prototype for _controlfp_s function */
#include <float.h>

unsigned int cw, orig_cw;
int err;

/* Get original FP control word and save it. */

err = _controlfp_s(&orig_cw, 0, 0);
if ( err ) /* handle error here */;

/* Set precision in FPU control word to single precision.
   This reduces the latency of divide and square-root operations. */
```

```
err = _controlfp_s(&cw, _PC_24, MCW_PC);
if ( err ) /* handle error here */;

/* Restore original FPU control word */

err = _controlfp_s(&cw, orig_cw, MCW_PC);
if ( err ) /* handle error here */;
```

2.22 Speeding Up Branches Based on Comparisons Between Floats

Optimization

Store operands of type float into a memory location and use integer comparison with the memory location to perform fast branches in cases where compilers do not support fast floating-point comparison instructions.

Application

This optimization applies to 32-bit software.

Rationale

Branches based on floating-point comparisons are often slow. AMD Family 10h and 12h processors support the FCOMI, FUCOMI, FCOMIP, and FUCOMIP instructions that allow implementation of fast branches based on comparisons between operands of type `double` or type `float`. However, some compilers do not support generating these instructions.

Some compilers only implement branches based on floating-point comparisons by using FCOM or FCOMP to compare the floating-point operands, followed by `FSTSW AX` in order to transfer the x87 condition-code flags into EAX. The subsequent branch is then based on the contents of the EAX register. Although the AMD Family 10h and 12h processors have acceleration hardware to speed up the FSTSW instruction, this process is still fairly slow.

Branches Dependent on Integer Comparisons Are Fast

One alternative for branches dependent upon the outcome of the comparison of operands of type `float` is to store the operand(s) into a memory location and then perform an integer comparison with that memory location. Branches dependent on integer comparisons are very fast. The replacement code uses a load dependent on an immediately prior store. If the store is not doubleword-aligned, no store-to-load-forwarding takes place, and the branch is still slow. Also, if there is a lot of activity in the load-store queue, forwarding of the store data may be somewhat delayed, thus negating some of the advantages of using the replacement code. It is recommended that you experiment with the replacement code to test whether it actually provides a performance increase in the code at hand.

The replacement code works well for comparisons against zero, including correct behavior when encountering a negative zero as allowed by the IEEE-754 standard. It also works well for comparing to positive constants. In that case, the user must first determine the integer representation of that floating-point constant. This can be accomplished with the following C code snippet:

```
float x;
scanf("%g", &x);
printf("%08X\n", (*((int *)(&x))));
```

The replacement code is IEEE-754 compliant for all classes of floating-point operands except NaNs.

Examples

Initial definitions:

```
#define FLOAT2INTCAST(f)  (*((int *)(&f)))
#define FLOAT2UINTCAST(f) (*((unsigned int *)(&f)))
```

Table 3. Comparisons Against Zero

Use this …	Instead of this.
`if (FLOAT2UINTCAST(f) > 0x80000000U)`	`if (f < 0.0f)`
`if (FLOAT2INCAST(f) <= 0)`	`if (f <= 0.0f)`
`if (FLOAT2INTCAST(f) > 0)`	`if (f > 0.0f)`
`if (FLOAT2UINTCAST(f) <= 0x80000000U)`	`if (f >= 0.0f)`

Table 4. Comparisons Against Positive Constant

Use this …	Instead of this.
`if (FLOAT2INTCAST(f) < 0x40400000)`	`if (f < 3.0f)`
`if (FLOAT2INTCAST(f) <= 0x40400000)`	`if (f <= 3.0f)`
`if (FLOAT2INTCAST(f) > 0x40400000)`	`if (f > 3.0f)`
`if (FLOAT2INTCAST(f) >= 0x40400000)`	`if (f >= 3.0f)`

Table 5. Comparisons Among Two Floats

Use this …	Instead of this.
`float t = f1 - f2;` `if (FLOAT2UINTCAST(t) > 0x80000000U)`	`if (f1 < f2)`
`float t = f1 - f2;` `if (FLOAT2INTCAST(t) <= 0)`	`if (f1 <= f2)`
`float t = f1 - f2;` `if (FLOAT2INTCAST(t) > 0)`	`if (f1 > f2)`
`float t = f1 - f2;` `f (FLOAT2UINTCAST(f) <= 0x80000000U)`	`if (f1 >= f2)`

2.23 Improving Performance in Linux® Libraries

Optimization

If symbol interposition is not important to a particular application, then you should control the visibility of the symbols in a shared object in such a way as to optimize internal references to other symbols in the library and minimize the symbol export table size.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Dynamically loadable libraries are a versatile feature of the Linux® operating system. These allow one or more symbols in one library to override an identical symbol in another library. Known as interposition, this ability makes customizations and probing seamless. Interposition is implemented by means of a procedure linkage table (PLT). The PLT is so flexible that even references to an overridden symbol inside its own library end up referencing the overriding symbol. However, the PLT imposes a performance penalty by requiring all function calls to public global routines to go through an extra step that increases the chances of cache misses and branch mispredictions. This is particularly severe for C++ classes whose methods refer to other methods in the same class.

When using `ld` to link a shared object, include the command line option `-Bsymbolic`.

If using a version of gcc prior to 4.0 to link a shared object, add the option `-Wl,-Bsymbolic` to the command-line. If using gcc 4.0 or later, add the option `-fvisibility=protected` to the command-line.

If finer control is desired, then it is possible to specify `-fvisibility=hidden` to gcc 4.0 or later and then add `__attribute__ ((visibility ("default")))` to each symbol that should be exported. When building C++ shared objects, also consider using the `-fvisibility-inlines-hidden` option.

2.24 Aligning Matrices

Optimization

When using multi-dimensional arrays or matrices, make sure that each row or 2nd-order dimension starts at a 16-byte boundary.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Instead of creating matrices with arbitrary dimensions, make sure that the size in bytes of the low-order dimension is a multiple of 16 and that it starts at a 16-byte boundary. By doing so, when iterating over the elements of the matrix the compiler is presented with data properly aligned for low-cost vectorization.

For example, in:

```
  double a [10][11],
         b [10][11];
  int i, j;

  for (j = 0; j < 10; j++)
    for (i = 0; i < 11; i++)
      b [j][i] = a [j][i] * M_1_PI;
```

Declare the matrices in this way:

```
__declspec (align (16))
       double a [10][ ((11 * sizeof (double) + 15) / 16) * 16 / sizeof (double)],
              b [10][ ((11 * sizeof (double) + 15) / 16) * 16 / sizeof (double)];
int i, j;

for (j = 0; j < 10; j++)
  for (i = 0; i < 11; i++)
    b [j][i] = a [j][i] * M_1_PI;
```

However, be aware of cache-bank conflicts for best performance. For more information, see section 5.7, "L1 Data Cache Bank Conflicts" on page 90.

Chapter 3 General 64-Bit Optimizations

The AMD x86-64 architecture provides a compatibility mode, which allows a 64-bit operating system to run existing 16-bit and 32-bit applications, and a 64-bit mode, which provides 64-bit addressing and expanded register resources to improve performance for recompiled 64-bit programs. This chapter presents general optimizations to improve the performance of software designed to run in 64-bit mode.

This chapter covers the following topics:

Topic	Page
64-Bit Registers and Integer Arithmetic	43
Using 64-bit Arithmetic for Large-Integer Multiplication	45
128-Bit Media Instructions and Floating-Point Operations	49
32-Bit Legacy GPRs and Small Unsigned Integers	49

3.1 64-Bit Registers and Integer Arithmetic

Optimization

Use 64-bit registers for 64-bit integer arithmetic.

Rationale

Using 64-bit registers instead of their 32-bit equivalents can dramatically reduce the amount of code necessary to perform 64-bit integer arithmetic.

Example 1

This code performs 64-bit addition using 32-bit registers:

```
; Add ECX:EBX to EDX:EAX, and place sum in EDX:EAX.
00000000  03 C3  add  eax, ebx
00000002  13 D1  adc  edx, ecx
```

Using 64-bit registers, the previous code can be replaced by one simple instruction (assuming that RAX and RBX contain the 64-bit integer values to add):

```
00000000  48 03 C3  add  rax, rbx
```

Although the preceding instruction requires one additional byte for the REX prefix, it is still one byte shorter than the original code. More importantly, this instruction still has a latency of only one cycle, uses two fewer registers, and occupies only one decode slot.

Example 2

To perform the low-order half of the product of two 64-bit integers using 32-bit registers, a procedure such as the following is necessary:

```
; In:         [ESP+8]:[ESP+4] = multiplicand
;             [ESP+16]:[ESP+12] = multiplier
; Out:        EDX:EAX = (multiplicand * multiplier) % 2^64
; Modifies: EAX, ECX, EDX, EFlags

llmul PROC
   mov edx, [esp+8]     ; multiplicand_hi
   mov ecx, [esp+16]    ; multiplier_hi
   or  edx, ecx         ; One operand >= 2^32?
   mov edx, [esp+12]    ; multiplier_lo
   mov eax, [esp+4]     ; multiplicand_lo
   jnz twomul           ; Yes, need two multiplies.
   mul edx              ; multiplicand_lo * multiplier_lo
   ret                  ; Done, return to caller.

twomul:
   imul edx, [esp+8]          ; p3_lo = multiplicand_hi * multiplier_lo
   imul ecx, eax              ; p2_lo = multiplier_hi * multiplicand_lo
   add  ecx, edx              ; p2_lo + p3_lo
   mul  dword ptr [esp+12]    ; p1 = multiplicand_lo * multiplier_lo
   add  edx, ecx              ; p1 + p2_lo + p3_lo = result in EDX:EAX
   ret                        ; Done, return to caller.

llmul ENDP
```

Using 64-bit registers, the entire product can be produced with only one instruction:

```
; Multiply RAX by RBX. The 128-bit product is stored in RDX:RAX.
00000000  48 F7 EB  imul  rbx
```

Related Information

For more examples of 64-bit arithmetic using only 32-bit registers, see the example on page 47 and “Efficient 64-Bit Integer Arithmetic in 32-Bit Mode” on page 130.

3.2 Using 64-bit Arithmetic for Large-Integer Multiplication

Optimization

Use 64-bit arithmetic for integer multiplication that produces 128-bit or larger products.

Background

Large integer multiplications (those involving 128-bit or larger products) are utilized in a variety of applications, such as cryptography software, which figure prominently in e-commerce applications and secure transactions on the Internet. Processors cannot perform large-number multiplication natively; they must break the operation into chunks that are permitted by their architecture (32-bit or 64-bit additions and multiplications).

Rationale

Using 64-bit rather than 32-bit integer operations dramatically reduces the number of additions and multiplications required to compute large products. For example, computing a 1024-bit product using 64-bit arithmetic requires fewer than one quarter the number of instructions required when using 32-bit operations:

Comparing...	32-bit arithmetic	64-bit arithmetic
Number of multiplications	256	64
Number of additions with carry	509	125
Number of additions	255	63

In addition, the processor performs 64-bit additions just as fast as it performs 32-bit additions, and the latency of 64-bit multiplications is only slightly higher than for 32-bit multiplications. (The processor is capable of performing three independent 64-bit additions each clock cycle and a 64-bit multiplication every other clock cycle.)

Example

Consider the multiplication of two unsigned 64-bit numbers *a* and *b*, represented in terms of 32-bit components *a1*:*a0* and *b1*:*b0*.

$a = a1 * 2^{32} + a0$

$b = b1 * 2^{32} + b0$

The product of *a* and *b*, calculated using the FOIL method of the polynomials above, can be expressed in terms of products of the 32-bit components, as follows:

Formula 3.1

$c = (a1 * b1) * 2^{64} + (a1 * b0 + a0 * b1) * 2^{32} + (a0 * b0)$

Each of the products of the components of *a* and *b* (for example, *a1 * b1*) is composed of 64 bits—an upper 32 bits and a lower 32 bits. It is convenient to represent these individual products as *d*, *e*, *f*, and *g*, as follows:

$a0 * b0 = d1{:}d0 = d1 * 2^{32} + d0$

$a1 * b0 = e1{:}e0 = e1 * 2^{32} + e0$

$a0 * b1 = f1{:}f0 = f1 * 2^{32} + f0$

$a1 * b1 = g1{:}g0 = g1 * 2^{32} + g0$

Substitution into Formula 3.1 above yields the following equation:

Formula 3.2

$c = (g1 * 2^{32} + g0) * 2^{64} + (e1 * 2^{32} + e0 + f1 * 2^{32} + f0) * 2^{32} + (d1 * 2^{32} + d0)$

Simplifying yields this equation:

Formula 3.3

$c = g1 * 2^{96} + (e1 + f1 + g0) * 2^{64} + (d1 + e0 + f0) * 2^{32} + d0$

It is convenient to represent the terms that are multiplied by each power of 2 as *c3*, *c2*, *c1*, and *c0*, as follows:

$g1 = c3$

$e1 + f1 + g0 = c2$

$d1 + e0 + f0 = c1$

$d0 = c0$

Substituting again yields:

Formula 3.4

$c = c3 * 2^{96} + c2 * 2^{64} + c1 * 2^{32} + c0$

The following procedure performs 64-bit unsigned integer multiplication, as previously illustrated using 32-bit integer operations:

```
; 32bitalu_64x64(int *a, int *b, int *c);
;
; TO ASSEMBLE INTO *.obj DO THE FOLLOWING:
;       ml.exe -coff -c 32bitalu_64x64.asm
;
.586
.K3D
.XMM
_DATA   SEGMENT
tempESP dd 0
_DATA   ENDS
_TEXT   SEGMENT
ASSUME DS:_DATA
PUBLIC _32bitalu_64x64
_32bitalu_64x64 PROC NEAR
;=============================================================================
; Save the register state. Registers EAX, ECX, and EDX are considered volatile
;  and assumed to be changed, while the registers below must be preserved.
push ebp
mov  ebp, esp
;=============================================================================
; Parameters passed into routine:
;  [ebp+8]   = ->a
;  [ebp+12]  = ->b
;  [ebp+16]  = ->c
;=============================================================================
push ebx
push esi
push edi
;=============================================================================
mov  esi,[ebp+8]      ; ESI = ->a
mov  edi,[ebp+12]     ; EDI = ->b
mov  ecx,[ebp+16]     ; ECX = ->c
push ebp
mov  [tempESP], esp
;=============================================================================
; Multiply 64-bit numbers a and b, each of which is composed of two 32-bit
;  components:
;  a = a1 * 2^32 + a0
;  b = b1 * 2^32 + b0
mov eax,[esi]      ; EAX = a0
mov edx,[edi]      ; EDX = b0
mul edx            ; EDX:EAX = a0*b0 = d1:d0
mov ebx,edx        ; EDX = d1
mov [ecx],eax      ; c0 = EAX
xor esp,esp        ; ESP = 0
xor ebp,ebp        ; EBP = 0
mov eax,[esi+4]    ; EAX = a1
mov edx,[edi]      ; EDX = b0
mul edx            ; EDX:EAX = a1*b0 = e1:e0
add ebx,eax        ; EBX = d1 + e0
adc ebp,edx        ; EBP = e1 + possible carry from d1+e0
adc esp,0          ; Collect possible carry into c3.
```

```
mov eax,[esi]       ; EAX = a0
mov edx,[edx+4]     ; EDX = b1
mul edx             ; EDX:EAX = a0*b1 = f1:f0
add ebx,eax         ; EBX = d1 + e0 + f0
adc ebp,edx         ; EBP = e1 + f1 + carry
adc esp,0           ; Collect possible carry into c3.
mov [ecx+4],ebx     ; c1 = d1 + e0 + f0

mov eax,[esi+4]     ; EAX = a1
mov edx,[edi+4]     ; EDX = b1
mul edx             ; EDX:EAX = a1*b1 = g1:g0
add ebp,eax         ; EBP = e1 + f1 + g0 + carry
adc esp,edx         ; ESP = g1 + carry
mov [ecx+8],ebp     ; c2 = e1 + f1 + g0 + carry
mov [ecx+12],esp    ; c3 = g1 + carry
;============================================================================
; Restore the register state.
mov esp, [tempESP]
pop ebp
pop edi
pop esi
pop ebx
mov esp, ebp
pop ebp
;============================================================================
ret
_32bitalu_64x64 ENDP
_TEXT   ENDS
END
```

To improve performance and substantially reduce code size, the following procedure performs the same 64-bit integer multiplication using 64-bit instead of 32-bit operations:

```
; 64bitalu_64x64(int *a, int *b, int *c);
;
; TO ASSEMBLE INTO *.obj DO THE FOLLOWING:
;       ml64.exe -c 64bitalu_64x64.asm
;
_TEXT   SEGMENT
64bitalu_64x64 PROC NEAR
;============================================================================
; Parameters passed into routine:
;  rcx = ->a
;  rdx = ->b
;  r8  = ->c
;============================================================================
mov rax, [rcx]     ; RAX = [a0]
mul [rdx]          ; Multiply [a0] by [b0] such that
                   ;  RDX:RAX = [c1]:[c0].
mov [r8], rax      ; Store 128-bit product of a and b.
mov [r8+8], rdx
;============================================================================
ret
64bitalu_64x64 ENDP
END
```

3.3 128-Bit Media Instructions and Floating-Point Operations

Optimization

Use 128-bit media (SSE, SSE2, SSE3, and SSE4a) instructions instead of x87 or 64-bit media instructions for floating-point operations.

Rationale

In 64-bit mode, the processor provides eight additional XMM registers (XMM8–XMM15) for a total of 16. These extra registers can substantially reduce register pressure in floating-point code written using 128-bit media instructions.

Although the processor fully supports the x87 and 64-bit media instructions, there are only eight registers available to these instructions (ST(0)–ST(7) or MMX0–MMX7, respectively). Additionally, the x87 and 64-bit media instructions require cumbersome register manipulation and mode switches, unlike SSE, SSE2, SSE3 and SSE4a.

For further information, see Chapter 9, “Optimizing with SIMD Instructions” on page 147.

3.4 32-Bit Legacy GPRs and Small Unsigned Integers

Optimization

Use the 32-bit legacy general-purpose registers (EAX through ESI) instead of their 64-bit extensions to store unsigned integer values whose range never requires more than 32 bits, even if subsequent statements use the 32-bit value in a 64-bit operation. (For example, use ECX instead of RCX until you need to perform a 64-bit operation; then use RCX.)

Rationale

In 64-bit mode, the machine-language representation of many instructions that operate on unsigned 64-bit register operands requires a REX prefix byte, which increases the size of the code. However, instructions that operate on a 32-bit legacy register operand do not require the prefix and have the desirable side-effect of clearing the upper 32 bits of the extended register to zero. For example, using the AND instruction on ECX clears the upper half of RCX.

Caution

Because the assembler also uses a REX prefix byte to encode the 32-bit sizes of the eight new 64-bit general-purpose registers (R8D–R15D), you should only use one of the original eight general-purpose registers (EAX through ESI) to implement this technique.

Example

The following example illustrates the unnecessary use of 64-bit registers to calculate the number of bytes remaining to be copied by an aligned block-copy routine after copying the first few bytes having addresses not meeting the routine's 8-byte-alignment requirements. The first two statements, after the program comments, use the 64-bit R10 register—presumably, because this value is later used to adjust a 64-bit value in R8—even though it requires no more than four bits to represent the range of values stored in R10. Using R10 instead of a smaller register requires a REX prefix byte (in this case, 49), which increases the size of the machine-language code.

```
; Input:
;  R10 = source address (src)
;  R8 = number of bytes to copy (count)
49 F7 DA      neg r10       ; Subtract the source address from 2^64.
49 83 E2 07   and r10, 7    ; Determine how many bytes were copied separately.
4D 2B C2      sub r8, r10   ; Subtract the number of bytes already copied from
                            ;  the number of bytes to copy.
```

To improve code density, the following rewritten code uses ECX until it is absolutely necessary to use RCX, eliminating two REX prefix bytes:

```
F7 D9      neg ecx       ; Subtract the source address from 2^32 (the processor
                         ; clears the high 32 bits of RCX).
83 E1 07   and ecx, 7    ; Determine how many bytes were copied separately.
4C 2B C1   sub r8, rcx   ; Subtract the number of bytes already copied from
                         ; the number of bytes to copy.
```

Chapter 4 Instruction-Decoding Optimizations

The optimizations in this chapter are designed to help maximize the number of instructions that the processor can decode at one time.

The AMD Family 10h and 12h processors instruction fetcher reads 32-byte packets from the L1 instruction cache. These packets are 32-byte aligned. The instruction bytes are then merged into a 32-byte pick window. On each cycle, the in-order front-end engine selects up to three AMD x86-64 instructions to decode from the pick window.

This chapter covers the following topics:

Topic	Page
DirectPath Instructions	52
Load-Execute Instructions for Floating-Point or Integer Operands	53
Loop Iteration Boundaries	56
32/64-Bit vs. 16-Bit Forms of the LEA Instruction	56
Take Advantage of x86 and AMD64 Complex Addressing Modes	57
Short Instruction Encodings	56
Stack Operations	59
Partial-Register Writes	60
Using LEAVE for Function Epilogues	64
Alternatives to SHLD Instruction	66
8-Bit Sign-Extended Immediate Values	67
8-Bit Sign-Extended Displacements	68
Code Padding with Operand-Size Override and Multibyte NOP	68

4.1 DirectPath Instructions

Optimization

❖ Use DirectPath instructions rather than VectorPath instructions. To determine the type of an instruction—either DirectPath or VectorPath—see Appendix C, “Instruction Latencies.”

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

DirectPath instructions minimize the number of operations per AMD64 instruction, thus providing for optimally efficient decode and execution. Up to three DirectPath Single instructions, or one and a half DirectPath Double instructions, can be decoded per cycle. VectorPath instructions block the decoding of DirectPath instructions.

The AMD Family 10h and 12h processors have been designed to execute the instructions most frequently generated by compilers as DirectPath Single or DirectPath Double instructions. However, assembly writers must still take into consideration the use of DirectPath versus VectorPath instructions.

Examples

The following example shows code for swapping two memory values. Although the second case uses an extra instruction, it is preferred because it avoids VectorPath instructions.

Avoid code such as the following which uses a VectorPath instruction.

```
   movzx eax, BYTE PTR [memA]
   xchg [memB], al    ; xchg mem8, reg8 is a VectorPath instruction
   mov [memA], al
```

Instead, use an equivalent instruction sequence such as the following using DirectPath instructions.

```
;; All of the following are DirectPath instructions
   movzx eax, BYTE PTR [memA]
   movzx ebx, BYTE PTR [memB]
   mov [memB], al
   mov [memA], bl
```

4.2 Load-Execute Instructions for Floating-Point or Integer Operands

A *load-execute instruction* is an instruction that loads a value from memory into a register and then performs an operation on that value. Many general purpose instructions, such as ADD, SUB, AND, etc., have load-execute forms:

```
ADD rax, QWORD PTR [foo]
```

This instruction loads the value `foo` from memory and then adds it to the value in the RAX register.

The work performed by a load-execute instruction can also be accomplished by using two discrete instructions—a load instruction followed by an execute instruction. The following example employs discrete load and execute stages:

```
MOV rbx, QWORD PTR [foo]
ADD rax, rbx
```

The first statement loads the value `foo` from memory into the RBX register. The second statement adds the value in RBX to the value in RAX.

The following optimizations govern the use of load-execute instructions:

- Load-Execute Integer Instructions on page 53.
- Load-Execute SSE/SSE2/SSE3 Instructions with Floating-Point or Integer Operands on page 54.
- Load-Execute x87 Instructions with Integer Operands on page 55.

4.2.1 Load-Execute Integer Instructions

Optimization

❖ When performing integer computations, use load-execute instructions instead of discrete load and execute instructions. Use discrete load and execute instructions only under one or more of the following circumstances:

- to explicitly schedule load and execute operations
- to avoid scheduler stalls for longer executing instructions
- if the load target will be used multiple times in different instructions

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Most load-execute integer instructions are DirectPath decodable and can be decoded at the rate of three per cycle. Splitting a load-execute integer instruction into two separate instructions reduces decoding bandwidth and increases register pressure, which results in lower performance.

4.2.2 Load-Execute SSE/SSE2/SSE3 Instructions with Floating-Point or Integer Operands

Optimization

❖ When performing floating-point computation using floating-point or integer source operands, use load-execute instructions instead of discrete load and execute instructions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Using load-execute floating-point instructions that take floating-point or integer operands improves performance for the following reasons:

- Denser code allows more work to be held in the instruction cache.
- Denser code generates fewer internal macro-ops, allowing the floating-point scheduler to hold more work, which increases the chances of extracting parallelism from the code.

The use of load-execute packed SSE instructions instead of distinct load and execute instructions improves performance in cases in which data might not be aligned on a 16-byte boundary. However, this requires setting the misaligned exception mask (MXCSR[17]). Setting this bit disables general protection exceptions for unaligned loads in SSE load-execute instructions. See also “Unaligned and Aligned Data Access” on page 149.

Example

Avoid code such as the following, which uses discrete load and execute SSE instructions:

```
movss xmm0, [float_var1]
movss xmm12, [float_var2]
mulss xmm0, xmm12
```

Instead, use code such as the following, which uses a load-execute SSE floating-point instruction:

```
movss xmm0, [float_var1]
mulss xmm0, [float_var2]
```

4.2.3 Load-Execute x87 Instructions with Integer Operands

Optimization

❖ Avoid x87 load-execute floating-point instructions that take integer operands (FIADD, FICOM, FICOMP, FIDIV, FIDIVR, FIMUL, FISUB, and FISUBR). When performing floating-point computations using integer source operands, use discrete load (FILD) and execute instructions instead.

However, when performing floating-point computations using floating point operands, use load-execute instructions instead of discrete load and execute instructions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The load-execute floating-point instructions that take integer operands take longer to execute than an FILD followed by the same register-execute arithmetic instruction. In some situations, this optimization can also reduce execution time, if FILD can be scheduled several instructions ahead of the arithmetic instruction to cover the FILD latency.

Example

Avoid code such as the following, which uses load-execute floating-point instructions that take integer operands:

```
fld   QWORD PTR [foo]    ; Push foo onto FP stack [ST(0) = foo].
fimul DWORD PTR [bar]    ; Multiply bar by ST(0) [ST(0) = bar * foo].
fiadd DWORD PTR [baz]    ; Add baz to ST(0) [ST(0) = baz + (bar * foo)].
```

Instead, use code such as the following, which uses discrete load and execute instructions:

```
fild  DWORD PTR [bar]    ; Push bar onto FP stack.
fild  DWORD PTR [baz]    ; Push baz onto FP stack.
fld   QWORD PTR [foo]    ; Push foo onto FP stack.
fmulp st(2), st          ; Multiply and pop [ST(1) = foo * bar, ST(0) = baz].
faddp st(1), st          ; Add and pop [ST(0) = baz + (foo * bar)].
```

4.3 Loop Iteration Boundaries

Optimization

Fit an entire loop iteration into the smallest number of 32-byte-aligned blocks. In program “hot spots” (as determined by either profiling or loop-nesting analysis), loop iteration boundaries should be placed at or near the beginning of code windows that are 32-byte aligned. The smaller the basic block, the more beneficial this optimization will be.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

AMD Family 10h and 12h processors can fetch 32 bytes of instruction data per cycle, but will not fetch across a 32-byte-aligned boundary. Aligning branch targets maximizes the number of instructions in the pick window and preserves instruction-cache space in branch-intensive code outside such hot spots.

4.4 32/64-Bit vs. 16-Bit Forms of the LEA Instruction

Optimization

Use the 32-bit or 64-bit forms of the Load Effective Address (LEA) instruction rather than the 16-bit form.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The 32-bit and 64-bit LEA instructions are implemented as DirectPath operations with an execution latency of only two cycles. The 16-bit LEA instruction, however, is a VectorPath instruction, which lowers the decode bandwidth and has a longer execution latency.

4.5 Take Advantage of x86 and AMD64 Complex Addressing Modes

Optimization

When porting from other architectures, remember that the x86 architecture provides many complex addressing modes. By building the effective address in one instruction, the instruction count can sometimes be reduced, leading to better code density and greater decode bandwidth. Refer to the section on effective addresses in the *AMD64 Architecture Programmer's Manual, Volume 1: Application Programming*, order# 24592 for more detailed information on how effective addresses are formed.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Building the effective address sometimes seems to require numerous instructions when there is a base address (such as the base of an array), an index and a displacement (if applicable). However, the x86 architecture can often handle all of this information in one instruction. This can reduce code size and results in fewer instructions to decode. As always, attention should be paid to total instruction length, latencies and whether or not the instruction choices are DirectPath (fastest) or VectorPath (slower).

Example

The first instruction sequence of five instructions having a total latency of 8 can be replaced by one instruction.

Number of Bytes	Latency	Instruction
3	1	`mov r11d, r10d`
8	2	`lea rcx, 68E35h`
3	1	`add r11, rcx`
5	3	`mov cl, BYTE PTR [r11+r13]`
2	1	`cmp cl, al`

The following instruction replaces the functionality of the above sequence.

Number of Bytes	Latency	Instruction
8	4	`cmp BYTE PTR [r10+r13+68E35h], al`

Example

These two instructions:

```
mov  r11, QWORD PTR ds:0x4c65a
mov  r11, QWORD PTR [r11+r8*8]
```

can be replaced by one instruction:

```
mov r11, QWORD PTR [r8*8+0x4c65a]
```

4.6 Short Instruction Encodings

Optimization

Use instruction forms with shorter encodings rather than those with longer encodings. For example, use 8-bit displacements instead of 32-bit displacements, and use the single-byte form of simple integer instructions instead of the 2-byte opcode-ModR/M form.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Using shorter instructions increases the number of instructions that can fit into the L1 instruction cache and increases the average decode rate.

Example

Avoid the use of instructions having longer encodings, such as the following:

```
81 C0 78 56 34 12  add eax, 12345678h  ; 2-byte opcode form (with ModRM)
81 C3 FB FF FF FF  add ebx, -5         ; 32-bit immediate value
0F 84 05 00 00 00  jz label1           ; 2-byte opcode, 32-bit immediate value
```

Instead, choose instructions having shorter encodings, such as:

```
05 78 56 34 12   add eax, 12345678h   ; 1-byte opcode form
83 C3 FB         add ebx, -5          ; 8-bit sign-extended immediate value
74 05            jz  label1           ; 1-byte opcode, 8-bit immediate value
```

4.7 Stack Operations

Optimization

When saving or restoring registers in function prologues or epilogues, or when passing arguments through the stack, use PUSH or POP instructions to improve performance and to reduce code size.

When deallocating stack space at function exit, use `RET imm` to improve performance and to reduce code size.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

In spite of the implicit dependency between several successive PUSH or POP instructions on the stack-pointer (which PUSH and POP modify), special circuitry (the Sideband Stack Optimizer) tracks the value that the stack-pointer assumes, allowing parallel execution of more than one PUSH or POP. This is also true of instructions that reference the stack-pointer, either implicitly or explicitly, including:

- Near CALL
- Near RET
- LEAVE
- Instructions that specify the stack-pointer as a source register.
- Instructions that specify the stack-pointer in the addressing mode of a memory operand without an index register.

However, the Sideband Stack Optimizer cannot break the dependency between the aforementioned instructions and other instructions that refer either implicitly or explicitly to the stack-pointer, including:

- LEA instructions that specify the stack-pointer.
- Instructions that specify the stack-pointer as a destination register.
- Instructions that specify the stack-pointer in the addressing mode of a memory operand with an index register.
- VectorPath instructions that specify the stack-pointer.

Examples

Avoid

```
sub   esp, 20
mov   [esp + 16], edi
mov   [esp + 12], esi
...
mov   [esp], eax
call  ...
...
mov   esi, [esp + 12]
mov   edi, [esp + 16]
add   esp, 20
ret
```

Preferred

```
sub   esp, 8
push  edi
push  esi
...
push  eax
call  ...
...
pop   esi
pop   edi
add   esp, 8
ret
```

4.8 Partial-Register Writes

Optimization

When writing to a register for the purpose of initialization, avoid instructions that

- write less than 32 bits of a general purpose integer register.
- write less than 128 bits of an XMM register.

When writing to a register during the course of a *non-initializing* operation on the register,

- avoid partial register writes.
- schedule any prior operations on the target register well ahead of the point in the code where the partial write is to occur.

Application

This optimization applies to:

- 32-bit software

- 64-bit software

Rationale

In order to handle partial register writes, the processor's execution core implements a data merging scheme. In the execution unit, an instruction that writes part of a register merges the modified portion with the current state of the other part of the register. This creates a false dependency on the most recent instruction that writes to any part of the register.

When writing to a register for the purpose of register initialization, it is usually possible to avoid false dependencies by careful instruction selection. For example, rather than initializing a part of a floating-point 128-bit XMM register, initialize the whole 128-bit register.

When writing to a register during the course of a non-initializing operation on the register, there is usually no additional performance loss due to partial register reads and writes. This is because, in the typical case, the partial register being written to is also a source operand to the operation.

For example, the following instruction does not suffer from merge dependencies:

```
addsd, xmm1, xmm2
```

However, in some cases of non-initializing operations on a register, it is preferable to avoid partial register writes and replace them with more efficient operations. In these cases, the partial register being written by the operation is not a source for the operation. Examples are provided below.

If it is not possible to avoid writing to a part of that register, you should schedule any such prior operation on any part of the register well ahead of the point where the partial write occurs. Such cases are also listed in the examples.

A general purpose integer register is viewed as a 64-bit register internal to the processor. A floating-point XMM register is viewed as one 128-bit register internal to the processor.

Current generation processors *cannot* write a 64-bit half of a 128-bit XMM register without having a merge dependency on the other 64-bit half. Additionally, current generation processors cannot write a 32-bit portion of a 128-bit XMM register without having a merge dependency on other bits of that register.

However, there are several instructions that initialize the lower 64 bits or 32 bits of an XMM register that also zero out the upper 64 or 96 bits and, thus, do not suffer from merge dependencies. For example, the following instructions do not have merge dependencies:

```
movsd xmm, [mem64]
movss xmm, [mem32]
```

Integer operations that write to the lower 32 bits of a general purpose integer register do not have a false merge dependency because they zero out the upper 32 bits. But operations that write to portions of a general purpose integer register narrower than 32 bits should be avoided.

Example 1

Avoid

```
MOV    al, bl
```

Preferred

```
MOVZX  eax, bl
```

Example 2

Avoid

```
MOV    al, [ebx]
```

Preferred

```
MOVZX  eax, byte ptr [ebx]
```

Example 3

Avoid

```
MOV    al, 01h
```

Preferred

```
MOV    eax, 00000001h
```

Example 4

The following recommendation only applies when both instruction operands of the `MOVSS` instruction are registers.

Avoid

```
MOVSS    xmm1, xmm2
```

Preferred

```
MOVAPS   xmm1, xmm2
```

Example 5

While this example uses instructions with double-precision data type, the principle also applies to instructions with single-precision data types.

Avoid

```
MOVLPD xmm1, QWORD PTR[eax]  ;      ; Address may or may not be
                                    ; 16-byte aligned.
MOVHPD xmm1, QWORD PTR[eax+8];
```

Preferred

```
MOVUPD xmm1, XMMWORD PTR[eax] ;
```

Example 6

Avoid

```
MOVLPD xmm1, QWORD PTR [eax]

MOVHPD xmm1, QWORD PTR [eax]     ; Same memory location as used for MOVLPD.
```

Preferred

```
MOVDDUP xmm1, QWORD PTR [eax];
```

Example 7

The following recommendation only applies when both operands of the MOVSD instruction are registers. If the source operand of the MOVSD instruction is a memory operand, the high-order 64 bits of the destination register are zeroed out, thereby avoiding any merge dependency on the destination register.

Avoid

```
MOVSD xmm1, xmm2
```

Preferred

```
MOVAPD xmm1, xmm2
```

Example 8

When the source operand of the MOVSD instruction is a memory operand, the high-order 64 bits of the destination register are zeroed out, thereby avoiding any merge dependency on the destination register.

Avoid

```
MOVLPD xmm1, mem64
```

Preferred

```
MOVSD xmm1, mem64
```

Example 9

The SQRTSD instruction writes only the lower 64 bits of xmm1 and so should be moved well below the write from the MULPD. In addition to SQRTSD, this example also applies for other instructions such as CVTPI2PS, CVTSI2SD, CVTSS2SD, UNPCKLPD, PUNPCKLQDQ, MOVLHPS and MOVHLPS.

Avoid

```
MULPD xmm1, xmm3
SQRTSD xmm1, xmm2
```

Preferred

```
MULPD xmm1, xmm3

.....schedule some other unrelated instructions here

SQRTSD xmm1, xmm2
```

4.9 Using LEAVE for Function Epilogues

Optimization

The recommended optimization for function epilogues depends on whether the function allocates local variables.

If the function	Then
Allocates local variables.	Replace the traditional function epilogue with the LEAVE instruction.
Does not allocate local variables or does not have a frame-pointer.	Do not use function prologues or epilogues. Access function arguments and local variables through rSP.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Functions That Allocate Local Variables

The LEAVE instruction is a single-byte instruction and saves 2 bytes of code space over the traditional epilogue. Replacing the traditional sequence with LEAVE also preserves decode bandwidth.

Functions That Do Not Allocate Local Variables

Accessing function arguments and local variables directly through ESP frees EBP for use as a general-purpose register.

Background

The function arguments and local variables inside a function are referenced through a so-called frame pointer. In AMD64 code, the base pointer register (rBP) is customarily used as a frame pointer. You set up the frame pointer at the beginning of the function using a function prologue:

```
push ebp                    ; Save old frame pointer.
mov  ebp, esp               ; Initialize new frame pointer.
sub  esp, n                 ; Allocate space for local variables (only if the
                            ; function allocates local variables).
```

Function arguments on the stack can now be accessed at positive offsets relative to rBP, and local variables are accessible at negative offsets relative to rBP.

Example

The traditional function epilogue looks like this:

```
mov esp, ebp    ; Deallocate local variables (only if space was allocated).
pop ebp         ; Restore old frame pointer.
```

Replace the traditional function epilogue with a single LEAVE instruction:

```
leave
```

4.10 Alternatives to SHLD Instruction

Optimization

Where register pressure is low, replace the SHLD instruction with alternative code using ADD and ADC, or SHR and LEA.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Using alternative code in place of SHLD achieves lower overall latency and requires fewer execution resources. The 32-bit and 64-bit forms of ADD, ADC, SHR, and LEA (except 16-bit form) are DirectPath instructions, while SHLD is a VectorPath instruction. Use of the replacement code optimizes decode bandwidth because it potentially enables the simultaneous decoding of a third DirectPath instruction. However, the replacement code may increase register pressure because it destroys the contents of one register (*reg2* in the following examples) whereas the register is preserved by SHLD.

Example 1

Replace this instruction:

```
shld reg32a, reg32b, 1 ; Operands are 32-bit registers.
```

with this code sequence:

```
add reg32b, reg32b ; Operands are 32-bit registers
adc reg32a, reg32a
```

Example 2

Replace this instruction:

```
shld reg1, reg2, 2
```

with this code sequence:

```
shr reg2, 30
lea reg1, [reg1*4+reg2]
```

Example 3

Replace this instruction:

```
shld reg1, reg2, 3
```

with this code sequence:

```
shr reg2, 29
lea reg1, [reg1*8+reg2]
```

4.11 8-Bit Sign-Extended Immediate Values

Optimization

Use 8-bit sign-extended immediate values instead of larger-size values.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Using 8-bit sign-extended immediate values improves code density with no negative effects on the processor.

Example

Consider this instruction:

```
add bx, -5
```

Avoid encoding it as:

```
81 C3 FF FB
```

Instead, encode it as:

```
83 C3 FB
```

4.12 8-Bit Sign-Extended Displacements

Optimization

Use 8-bit sign-extended displacements for conditional branches.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Using short, 8-bit sign-extended displacements for conditional branches improves code density with no negative effects on the processor. See “Branch Alignment and Density” on page 99 for more details on optimizing branches.

4.13 Code Padding with Operand-Size Override and Multibyte NOP

Optimization

Use the multibyte NOP instruction (0F 1Fh) to align code and space out branches..

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Occasionally it is necessary to insert neutral code fillers into the code stream (for example, for code-alignment purposes or to space out branches). Because this filler code is executable, it should take up as few execution resources as possible, should not diminish decode density, and should not modify any processor state other than to advance the instruction pointer (rIP). Although there are several possible multibyte NOP-equivalent instructions that do not change the processor state (other than rIP), combinations of the operand-size override and the multibyte NOP instruction are more efficient. These NOP instructions are only available on AMD Athlon™ and later processors. For processors older than Athlon, use the standard NOP (opcode 090h) in combination with up to three operand size

override prefixes (opcode 66h). The use of more than three legacy prefixes limits decoder performance.

Example

Assign code-padding sequences like these and use them to align code and space out branches. These sequences are suitable for both 32-bit and 64-bit code, and you can use them on the AMD Family 10h and 12h processors:

```
NOP1_OVERRIDE_NOP TEXTEQU <DB 090h>
NOP2_OVERRIDE_NOP TEXTEQU <DB 066h, 090h>
NOP3_OVERRIDE_NOP TEXTEQU <DB 00fh, 01fh, 000h>
NOP4_OVERRIDE_NOP TEXTEQU <DB 00fh, 01fh, 040h, 000h>
NOP5_OVERRIDE_NOP TEXTEQU <DB 00fh, 01fh, 044h, 000h, 000h>
NOP6_OVERRIDE_NOP TEXTEQU <DB 066h, 00fh, 01fh, 044h, 000h, 000h>
NOP7_OVERRIDE_NOP TEXTEQU <DB 00fh, 01fh, 080h, 000h, 000h, 000h, 000h>
NOP8_OVERRIDE_NOP TEXTEQU <DB 00fh, 01fh, 084h, 000h, 000h, 000h, 000h, 000h>
NOP9_OVERRIDE_NOP TEXTEQU <DB 066h, 00fh, 01fh, 084h, 000h, 000h, 000h, 000h,
000h>
NOP10_OVERRIDE_NOP TEXTEQU <DB 066h, 066h, 00fh, 01fh, 084h, 000h, 000h, 000h,
000h, 000h>
NOP11_OVERRIDE_NOP TEXTEQU <DB 066h, 066h, 066h, 00fh, 01fh, 084h, 000h, 000h,
000h, 000h, 000h>
```

In certain rare situations, padding of up to 31 bytes can improve performance by aligning “hot” branch targets. For example, run-time profile information may reveal that a forward branch is very often taken. In these cases, generate padding by combining a minimum number of the large NOP instructions used in the above code-padding sequences.

For x87 floating-point instructions, a better single-byte padding exists. See “Align and Pack DirectPath x87 Instructions” on page 177. For aligning loop tops, see “Loop Iteration Boundaries” on page 56.

Chapter 5 Cache and Memory Optimizations

The optimizations in this chapter take advantage of the large L1 caches and high-bandwidth buses of AMD Family 10h and 12h processors.

This chapter covers the following topics:

Topic	Page
Memory-Size Mismatches	71
Natural Alignment of Data Objects	73
Store-to-Load Forwarding Restrictions	74
Prefetch and Streaming Instructions	81
Write-Combining	89
L1 Data Cache Bank Conflicts	90
Placing Code and Data in the Same 64-Byte Cache Line	91
Memory and String Routines	92
Stack Considerations	94
Cache Issues When Writing Instruction Bytes to Memory	95
Interleave Loads and Stores	96
Using 1-Gbyte Virtual Memory Pages	96

5.1 Memory-Size Mismatches

Optimization

❖ Avoid memory-size mismatches when different instructions operate on the same data. When one instruction stores and another instruction subsequently loads the same data, align instruction operands and keep the loads/stores of each operand the same size.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Examples—Store-to-Load-Forwarding Stalls

The following code examples result in a store-to-load-forwarding stall:

Avoid (64-bit)

```
foo DQ ?                    ; Assume foo is 8-byte aligned.
...
mov DWORD PTR foo, eax      ; Store a DWORD to foo.
mov DWORD PTR foo+4, ebx    ; Now store to foo+4.
mov rcx, QWORD PTR foo      ; Load a QWORD from foo.
```

Avoid (32-bit)

```
foo DQ ?                    ; Assume foo is 4-byte aligned.
...
mov DWORD PTR foo, eax      ; Store a DWORD in foo.
mov DWORD PTR foo+4, edx    ; Store a DWORD in foo+4.
fld QWORD PTR foo           ; Load a QWORD from foo.
```

Avoid

```
mov  foo, eax
mov  foo+4, edx
...
movq mm0, foo
```

Preferred

```
mov       foo, eax
mov       foo+4, edx
...
movd      mm0, foo
punpckldq mm0, foo+4
```

Preferred If Stores Are Close to the Load

```
movd      mm0, eax
mov       foo+4, edx
punpckldq mm0, foo+4
```

Examples—Large-to-Small Mismatches

Avoid large-to-small mismatches, as shown in the following code:

Avoid (64-bit)

```
foo DQ ?                    ; Assume foo is 8-byte aligned.
...
mov QWORD PTR foo, rax      ; Store a QWORD to foo.
mov eax, DWORD PTR foo      ; Load a DWORD from foo.
mov edx, DWORD PTR foo+4    ; Load a DWORD from foo+4.
```

Avoid (32-bit)

```
foo DQ ?                    ; Assume foo is 4-byte aligned.
```

```
...
fst QWORD PTR foo           ; Store a QWORD in foo.
mov eax, DWORD PTR foo      ; Load a DWORD from foo.
mov edx, DWORD PTR foo+4    ; Load a DWORD from foo+4.
```

Avoid

```
movq foo, mm0
...
mov  eax, foo
mov  edx, foo+4
```

Preferred

```
movd   foo, mm0
pswapd mm0, mm0
movd   foo+4, mm0
pswapd mm0, mm0
...
mov    eax, foo
mov    edx, foo+4
```

Preferred If the Contents of MM0 are No Longer Needed

```
movd      foo, mm0
punpckhdq mm0, mm0
movd      foo+4, mm0
...
mov       eax, foo
mov       edx, foo+4
```

Preferred If the Stores and Loads are Close Together, Option 1

```
movd   eax, mm0
pswapd mm0, mm0
movd   edx, mm0
pswapd mm0, mm0
```

Preferred If the Stores and Loads are Close Together, Option 2

```
movd      eax, mm0
punpckhdq mm0, mm0
movd      edx, mm0
```

5.2 Natural Alignment of Data Objects

Optimization

❖ Make sure data objects are *naturally aligned*. An object is naturally aligned if it is located at an address that is a multiple of its size.

Locate this type of object	At an address evenly divisible by
Word	2
Doubleword	4
Quadword	8
Ten-byte (for example, TBYTE or REAL10)	8 (instead of 10)
Double quadword	16

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

A misaligned store or load operation suffers a minimum one-cycle penalty in the processor's load-store pipeline. Also, using misaligned loads and stores increase the likelihood of encountering a store-to-load forwarding pitfall, especially when operating in long mode (64-bit software). (For a more detailed discussion of store-to-load forwarding issues, see "Store-to-Load Forwarding Restrictions" on page 74.)

In addition, if the Alignment Mask bit is set in Control Register 0 (CR0), an unaligned memory reference may cause an alignment check exception. For more information on this topic, see the *AMD64 Architecture Programmer's Manual, Volume 2,* order# 24593.

5.3 Store-to-Load Forwarding Restrictions

Optimization

Maintain consistent operand sizes across all loads and stores. Preferably use doubleword, quadword, or 128-bit operand sizes. Avoid store-to-load forwarding pitfalls, such as

- narrow-to-wide forwarding cases.
- mismatched addresses for stores and loads.
- misaligned data references.
- loading data from anywhere in the same doubleword of memory other than the identical start addresses of the stores when using word or byte stores

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Store-to-load forwarding refers to the process of a load reading (forwarding) data from the store buffer. Where this is possible, it can lead to a performance improvement because the load does not have to wait for the recently written (stored) data to be written to cache and then read back out again.

There are circumstances under which AMD Family 10h and 12h processor load-store (LS) architecture does not allow data to be read from a store in the store buffer. In these cases, it is impossible to load the needed data into a register until the store has retired out of the store buffer and written to the data cache. A store-buffer entry cannot retire and write to the data cache until *every* instruction before the store has completed and retired from the reorder buffer. The implication of this restriction is that all instructions in the reorder buffer, up to and including the store, must complete and retire out of the reorder buffer before the load can complete. Effectively, the load has a false dependency on every instruction up to the store.

Due to the significant depth of the LS buffer of AMD Family 10h and 12h processors, any load that is dependent on a store that cannot bypass data through the LS buffer may experience significant delays of up to tens of clock cycles, where the exact delay is a function of pipeline conditions.

The following sections describe store-to-load forwarding examples.

Store-to-Load Forwarding Pitfalls—True Dependencies

A load is *not* allowed to read data from the store-buffer entry if any of the following conditions occur:

- The start address of the load does not match the start address of the store.
- The load operand size is greater than the store operand size.
- Either the load or the store is misaligned. See “Natural Alignment of Data Objects” on page 73 for additional information on alignment recommendations.
- A high byte (or word) store and a low byte (or word) store in the same aligned doubleword are followed by either a low or high byte (or word) load.

The following sections describe common-case scenarios to avoid. In these scenarios, a load has a true dependency on an LS2-buffered store, but cannot read (forward) data from a store-buffer entry.

Load Operand Size Greater than the Store Operand Size

If the following conditions are present, there is a narrow-to-wide store-buffer data-forwarding restriction:

- The operand size of the store data is smaller than the operand size of the load data.
- The range of addresses spanned by the store data covers some subrange of the addresses spanned by the load data.

Examples

Avoid

```
mov eax, 10h
mov WORD PTR [eax], bx     ; Word store
...
mov ecx, DWORD PTR [eax]   ; Doubleword load--cannot forward upper byte
                           ; from store buffer
```

Avoid

```
MOV eax, 10h
MOV BYTE PTR [eax+3], bl   ; Byte store
...
MOV ecx, DWORD PTR [eax]   ; Doubleword load--cannot forward upper byte
                           ; from store buffer
```

Avoid

```
MOV eax, 10h
MOVSD QWORD PTR [eax], xmm0 ; Quadword store
MOVSD QWORD PTR [eax+8], xmm1 ; Quadword store

...

MOVAPD xmm2, XMMWORD PTR [eax] ; Octal Word load--cannot forward upper and
                               ; lower Quadwords from store buffer.
```

Preferred

```
MOV eax, 10h

MOVAPD xmm3, xmm0 ; Assumes XMM3 is available and will not be detrimental
                  ; to register pressure

SHUFPD xmm3, xmm1, 0

MOVAPD XMMWORD PTR [eax], xmm3

....

MOVAPD xmm2, XMMWORD PTR [eax] ; Octal Word load--can forward from
                               ; Octal word store from store buffer
```

Mismatched Store and Load Addresses

A data-forwarding restriction exists if the start address of the store data does not match the start address of the load data.

In general, wide stores can forward data to narrow loads if the start address of the load matches that of the store and neither store nor load is misaligned. However, store-to-load forwarding cannot occur if the start addresses of the load and store do not match, with the exception that stores to an aligned 128-bit location can forward to loads of 64 bits or less, starting at its upper 64-bit quadword.

Examples

Avoid

```
movq [foo], mm1      ; Store upper and lower half.
...
add  eax, [foo]      ; Fine
add  edx, [foo+4]    ; Not good!
```

Preferred

```
movd      [foo], mm1     ; Store lower half.
punpckhdq mm1, mm1       ; Copy upper half into lower half.
movd      [foo+4], mm1   ; Store lower half.
...
add       eax, [foo]     ; Fine
add       edx, [foo+4]   ; Fine
```

Acceptable

```
mov eax, 10h
movapd XMMWORD PTR [eax], xmm0 ; Store upper and lower half.

...

movsd xmm1, QWORD PTR [eax] ; Fine
movsd xmm2, QWORD PTR [eax+8] ; Load of upper 64 bits, OK.
```

Misaligned Store-Buffer Data-Forwarding Restriction

If the following condition is present, there is a misaligned store-buffer data-forwarding restriction:

- The store or load address is misaligned. For example, a quadword store is not aligned to a quadword boundary.

A common case of misaligned store-data forwarding involves the passing of misaligned quadword floating-point data on the doubleword-aligned integer stack. Avoid the type of code shown in the following example:

```
mov  esp, 24h
fstp QWORD PTR [esp]    ; ESP = 24
```

```
...                     ; Store occurs to quadword misaligned address.
fld  QWORD PTR [esp]    ; Quadword load cannot forward from quadword
                        ;  misaligned 'FSTP[ESP]' store operation.
```

Forwarding Restriction from Distinct Stores on Distinct Bytes (or Words) to a Subsequent Load on One of the Same Bytes (or Words) Within the Same Aligned Doubleword Location

When there are two or more distinct stores to distinct bytes (or words) inside the same aligned doubleword memory location, it may not be possible to forward the data from the stores to a subsequent load from one of the same byte (or word) locations. Therefore, it is recommended to use doubleword, quad word or 128-bit operand sizes to allow store-to-load forwarding.

However, when there is only a single store and a single load to a byte (or word) inside an aligned doubleword location, store-to-load forwarding is allowed to occur as long as all of the other conditions listed previously in this section for store-to-load forwarding are satisfied.

Examples

In all of the examples below, the operations in between the store and the load indicated by the ... are assumed not to write to any part of the aligned doubleword under consideration.

Allowed

```
mov eax, 10h
mov BYTE PTR [eax], bl ; Low-byte store to an aligned doubleword
...
mov dl, BYTE PTR[eax] ; low byte load CAN forward from low byte store
```

Allowed

```
mov eax, 10h
mov WORD PTR [eax], bx ; Low-word store to an aligned doubleword
...

mov dl, BYTE PTR[eax] ; low byte load CAN forward from low word store
```

Allowed

```
mov eax, 10h
mov DWORD PTR [eax], ebx ; doubleword store to an aligned doubleword
...
mov dx, WORD PTR[eax] ; low word load CAN forward from doubleword store
```

Avoid

```
mov eax, 10h
mov BYTE PTR[eax], bl  ; Low-byte store to an aligned doubleword
mov BYTE PTR[eax+1], bh ; High-byte store to an aligned doubleword
```

```
...
mov dl, BYTE PTR [eax]  ; low byte load cannot forward from low byte store
```

Preferred

```
mov eax, 10h
mov WORD PTR [eax], bx ;
...
mov dl, BYTE PTR [eax] ;
```

Avoid

```
mov eax, 10h
mov BYTE PTR[eax], bl  ; Low-byte store to an aligned doubleword
mov BYTE PTR[eax+1], bh ; High-byte store to an aligned doubleword
...
mov dl, BYTE PTR [eax]  ; low byte load cannot forward from low byte store
mov dh, BYTE PTR [eax+1]; high byte load CAN forward from high byte store
```

Preferred

```
mov eax, 10h
mov WORD PTR [eax], bx ;
...
mov dx, WORD PTR [eax] ;
```

Summary of Store-to-Load-Forwarding Pitfalls to Avoid

The following list summarizes the situations that require care to handle store-to-load forwarding cases:

- Avoid narrow-to-wide forwarding cases.
- Avoid mismatched addresses for stores and loads.
- Avoid misaligned data references.
- When using word or byte stores, avoid having two or more distinct stores to distinct bytes (or words) inside the same aligned doubleword memory location followed by a subsequent load from one of the same byte (or word) locations.
- Maintain consistent operand sizes across all loads and stores. Preferably use doubleword, quadword, or 128-bit operand sizes.

5.4 Good Practices for Avoiding False Store-to-Load Forwarding

Optimization

Choose linear addresses for the source and destination operands of REP MOVS/CMPS that are not an exact multiple of 4K pages away from each other.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

As mentioned in the previous section, store-to-load forwarding occurs when the store address matches the load address. This address match is split into two stages. In the first stage, bits 4:11 of the store and the load addresses are matched. In addition the double word mask of the store and load addresses is matched. The double word mask indicates whether the load/store pair is accessing the same double word in a 16-byte bank. If both these parameters match, then a store-to-load forward is initiated. In the second stage the remaining bits 12:47 of the store and load addresses is matched. If the remaining bits match, then the STLF is considered as a true STLF and is allowed to proceed. Otherwise it is considered as a false STLF and the load is cancelled and retried.

The previous section deals with true STLF and describes the practices to follow to promote it. This section deals with the cases of *false* STLF and what the developer needs to do to avoid these from occurring in the first place, thereby avoiding the later penalty of STLF cancellation.

Example

For REP MOVS/CMPS, choose linear addresses that avoid conflicts.

REP stands for the repeat function. This function repeats or iterates its associated string instruction as many times as specified in the counter register (rCX) and terminates the repetition when the value in rCX reaches 0. For example, REP MOVS moves a string from a source address to a destination address a specified number of times. In the event that bits 4:11 of the linear address of the store address in the first iteration match the load address in the second iteration, a store-to-load forward may be initiated.

When the destination address of an iteration is located at an exact multiple of 4K pages away from the source address of the next iteration, an STLF will be initiated. When the remaining address bits are found to be mismatched later, the STLF is cancelled and the load has to be retried. This results in a significant penalty of wasted DC bandwidth due to having to retry loads multiple times.

For example, a REP MOVS instruction suffers from these inefficiencies if RSI is 0x1ffeee000000 and RDI is 0x1ffeee401000.

5.5 Prefetch and Streaming Instructions

Optimization

❖ Where appropriate, use one of the prefetch instructions to increase the effective bandwidth of AMD Family 10h and 12h processors.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Prefetch instructions take advantage of the high bus bandwidth of AMD Family 10h and 12h processors to hide latencies when fetching data from system memory. A prefetch instruction initiates a read request of a specified address and reads the entire cache line that contains that address.

AMD Family 10h and 12h processors perform three types of prefetches:

Prefetch type	Description
Load	Reads the data into the L1 data cache; the data is later evicted to the L2 cache. The following instructions perform load prefetches: PREFETCH, PREFETCHT0, PREFETCHT1, and PREFETCHT2.
Store	Reads the data into the L1 data cache and marks the data as modified; the data is later evicted to the L2 cache. The PREFETCHW instruction performs a store prefetch.
Nontemporal	The PREFETCHNTA instruction performs a nontemporal prefetch. The data is read into the L1 data cache; to avoid cache pollution, when a PREFETCHNTA misses in the L2 cache and reads from memory, the data is never evicted to the L2 cache. When a PREFETCHNTA hits in the L2 cache, the data is evicted back to the L2 cache.

The prefetch instructions can be used anywhere, in any type of code. The use of prefetch instructions is not affected by the values of Control Register 0 (CR0) bits, such as CR0.EM and CR0.TS.

Prefetching versus Preloading

In code that makes irregular memory accesses rather than sequential accesses, an ordinary MOV instruction is the best way to load data. But in situations where sequential addresses are read, prefetch instructions can improve performance. Prefetch instructions only update the L1 data cache and do not update an architectural register.

Unit-Stride Access

Large data sets typically require unit-stride access to ensure that all data pulled in by a prefetch instruction are actually used. Large data sets make use of all data that are read from memory, rather than using only a sparse subset of the memory. If necessary, you should reorganize algorithms or data structures to allow unit-stride access. For a definition of unit-stride access, see “Definitions” on page 88.

Hardware Prefetcher Optimizations

Previous AMD64 processors prefetched data into the L2 cache. In AMD Family 10h processors, the data hardware prefetcher loads data into the L1 cache. This hides the L2 cache access latency and offers significant performance improvement. However, this slightly increases the risk of thrashing and cache pollution, as discussed later in Section 5.5.

The hardware prefetcher in AMD Family 10h processors is a unit-stride prefetcher which trains on L1 cache misses and propagates on L1 cache accesses (hits and misses). Any two consecutive cache line misses can train a hardware prefetch stream, and this generates a preset number of initial prefetch requests. Any subsequent unit-stride accesses propagate the stream, causing more cache lines to be prefetched. For example, if the preset value is 3 and there are L1 misses to lines I and I+1, the prefetcher issues prefetch requests for lines I+2, I+3, and I+4. Subsequently, if a hit or miss to line I+2 is seen, a prefetch request for I+5 is issued, and so on.

Adaptive Prefetching

In AMD Family 10h processors, a hardware optimization called adaptive prefetching is implemented to improve the timeliness of the prefetches. This mechanism basically kicks in if the demand stream catches up with the prefetch stream, and it adjusts the prefetch distance dynamically to maintain a good prefetch distance. Here, a good prefetch distance is defined as the number of cachelines the prefetch stream needs to stay ahead, such that the demand stream hits on prefetched lines in the L1 cache. Again, the hardware has a preset maximum fetch-ahead distance that controls this dynamic scheme. As part of this adaptive scheme, the hardware prefetcher in AMD Family 10h processors maintains pending prefetch request counters and adjustable distance counters. This helps the prefetcher to scale with different memory technologies.

Note that in AMD Family 10h processors, the hardware prefetcher trains on software prefetch requests (including the NTA type).

Contraindications for Prefetching

There are situations in which careless software prefetching can hurt performance.

- Thrashing—This is potentially the worst scenario. Thrashing occurs if more than two arrays are prefetched in parallel and the addresses are separated by whole multiples of 32K bytes (the L1 cache size divided by the associativity). When this occurs, some of the prefetched data evicts other prefetched data before it can be used. This is inefficient even without prefetching—which simply makes the situation worse. Thrashing can be particularly bad if PREFETCHNTA is used.

- Cache pollution—This is a problem when the code prefetches a large amount of unused data, such as when the data is used conditionally or consists of many short sequences and the prefetches extend beyond the ends of the ranges of addresses that are actually desired.
- Prefetch from unmapped pages—This occurs when there is a prefetch in a loop, and the prefetch address is simply the data address plus some offset. Normally you should make the offset large enough so the data is fetched before the loop catches up to it, but this means there will be some over-run at the end of the loop. An over-run in an unmapped page can result in a significant delay. This is not so important if the over-run falls at the end of a very long stream of useful data.

In general, prefetching is useful where the program is neither totally memory-bound nor totally compute-bound, and the pattern of data access is fairly predictable within the code. The ideal fetch-ahead distance depends on the code, on the DRAM latency, and on how the data is laid out in address space.

The following table summarizes which prefetch instructions to use based on data size and data type.

Table 6. Prefetching Guidelines

Data	Less Than ½ L1 Size	Less Than ½ L2 Size or Unknown Size		Greater Than ½ L2 Size
		Reused	Not Reused	
Read-Only	PREFETCH[1] or PREFETCHNTA	PREFETCH[1]	PREFETCHNTA	PREFETCHNTA
Sequential Read-Only	Prefetcher + PREFETCH[1,3]	Prefetcher + PREFETCH[1,3]	PREFETCHNTA	PREFETCHNTA
Read-Write	PREFETCHW	PREFETCHW	PREFETCHNTA	PREFETCHNTA
Sequential Read-Write	PREFETCHW	PREFETCHW	PREFETCHNTA	PREFETCHNTA
Write-Only	PREFETCHW	PREFETCHW	MOVNT[2,5]	MOVNT[2,5]
Sequential Write-Only	Prefetcher + PREFETCHW[4]	Prefetcher + PREFETCHW[4]	MOVNT[2,5]	MOVNT[2,5]

Notes:

1. *PREFETCH is a place-holder for any of PREFETCH, PREFETCHT0, PREFETCHT1 or PREFETCHT2.*
2. *MOVNT is a place-holder for any of MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPD, MOVNTPS, MOVNTSD, MOVNTSS, MASKMOVQ or MASKMOVDQU.*
3. *Use PREFETCH[1] twice before iterations to jump-start the prefetcher, if advantageous. Otherwise, do not use PREFETCH[1].*
4. *Use PREFETCHW twice before iterations to jump-start the prefetcher, if advantageous. Otherwise, do not use PREFETCHW.*
5. *If no suitable MOVNT[2] instruction is available, use PREFETCHNTA.*

For guidance on when to use software prefetching for memory and string routines, see Section 5.9, “Memory and String Routines’ on page 92.

PREFETCH/W versus PREFETCHNTA/T0/T1/T2

PREFETCHNTA, PREFETCHT0, PREFETCHT1, and PREFETCHT2 are SSE instructions and are processor-implementation dependent. For AMD Family 10h and 12h processors, data that is prefetched with the PREFETCHNTA instruction is not placed into the L2 cache when it is evicted unless it was originally in L2 when prefetched.

PREFETCHNTA is intended for non-temporal data that will not be needed again soon. PREFETCHNTA should also be used when reading arrays that are so large that they are larger than the L2 cache. Because of their size, such large arrays will not be available in L2 even if they are needed again, and by feeding them through the L2 cache, other possibly useful data will also be evicted from L2.

Note: *The sizes of the L1 and L2 caches of the processor can be determined by using the CPUID instruction.*

Note: *DC misses on PREFETCHNTA trigger the hardware prefetcher on AMD Family 10h and 12h processors, but those prefetch streams are marked "NT", so that they are not evicted back to L2 or L3.*

Note: *PREFETCHNTA should* not *be used for large arrays that are only being written, not read. In such cases, write-combining stores should be used. (See "Write-Combining" on page 89, Appendix B "Implementation of Write-Combining" on page 239, and "Write-Combining" in the AMD64 Architecture Programmer's Manual, Volume 2, order# 24593.)*

AMD Family 10h and 12h processors implement the PREFETCHT0, PREFETCHT1, and PREFETCHT2 instructions in exactly the same way as the PREFETCH instruction. That is, the data is brought into the L1 data cache. This functionality could change in future implementations of the AMD Family 10h and 12h processors.

PREFETCHW versus PREFETCH

Code intended to modify the cache line that is brought in through prefetching should use the PREFETCHW instruction. PREFETCHW provides a hint to the AMD Family 10h and 12h processors of an intent to modify the cache line. The AMD Family 10h and 12h processors mark the cache line being read by PREFETCHW as *modified*. Using PREFETCHW can save additional cycles compared to PREFETCH, and avoid the subsequent cache state change caused by a write to the prefetched cache line. Only use PREFETCHW if there is a write to the same cache line afterwards.

Use of Streaming Instructions

Use streaming instructions instead of PREFETCHW in situations where all of the following conditions are true:

- The code will overwrite one or more complete cache lines with new data.
- The new data will not be used again soon.

Streaming instructions include the non-temporal stores MOVNTDQ, MOVNTI, MOVNTPS, MOVNTPD, MOVNTSD, MOVNTSS and the MMX instruction MOVNTQ. However, unlike regular stores, non-temporal stores are weakly ordered relative to other loads and stores. If strong ordering of stores is required, an SFENCE instruction should be used between the non-temporal stores and any succeeding normal stores. See Section 11.4, “Memory Barrier Operations’ on page 200 for further recommendations on memory barrier instructions.

Streaming instructions can dramatically improve memory-write performance. They write data directly to memory through write-combining buffers, bypassing the cache. This is faster than PREFETCHW because data does not need to be initially read from memory to fill the cache lines, only to be completely overwritten shortly thereafter. The new data is simply written to memory, replacing the old data in memory, so no memory read is performed.

One application where streaming is useful, often in conjunction with prefetch instructions, is in copying large blocks of memory.

Note: *The streaming instructions are not recommended or necessary for write-combined memory regions since the processor automatically combines writes for those regions. Write-combine memory types are indicated through the MTRRs and the page-attribute table (PAT).*

Note: *For best performance, do not mix streaming instructions on a cache line with non-streaming store instructions.*

For more information on write-combining, see Appendix B, “Implementation of Write-Combining.”

Multiple Prefetches

Programmers can initiate multiple outstanding prefetches on AMD Family 10h and 12h processors. These processors can have a theoretical maximum of eight outstanding cache misses, including prefetches. When all resources are filled by various memory read requests, the processor waits until resources become free before processing the next request. Multiple prefetch requests are essentially handled in order, prefetching data in the order that it is needed.

The following example shows how to initiate multiple prefetches when traversing more than one array.

Example—Multiple Prefetches

```
.CODE
.K3D
.686

; Original C code:
;
; #define LARGE_NUM 65536
; #define ARR_SIZE (LARGE_NUM*8)
;
; double array_a[LARGE_NUM];
; double array_b[LARGE_NUM];
; double array_c[LARGE_NUM];
```

```
; int i;
;
; for (i = 0; i < LARGE_NUM; i++) {
;    a[i] = b[i] * c[i];
; }

   mov edx, (-LARGE_NUM)     ; Use biased index.
   mov eax, OFFSET array_a   ; Get address of array_a.
   mov ebx, OFFSET array_b   ; Get address of array_b.
   mov ecx, OFFSET array_c   ; Get address of array_c.

loop:
   prefetchw [eax+256]   ; Four cache lines ahead
   prefetch  [ebx+256]   ; Four cache lines ahead
   prefetch  [ecx+256]   ; Four cache lines ahead
   fld  QWORD PTR [ebx+edx*8+ARR_SIZE]       ; b[i]
   fmul QWORD PTR [ecx+edx*8+ARR_SIZE]       ; b[i] * c[i]
   fstp QWORD PTR [eax+edx*8+ARR_SIZE]       ; a[i] = b[i] * c[i]
   fld  QWORD PTR [ebx+edx*8+ARR_SIZE+8]     ; b[i+1]
   fmul QWORD PTR [ecx+edx*8+ARR_SIZE+8]     ; b[i+1] * c[i+1]
   fstp QWORD PTR [eax+edx*8+ARR_SIZE+8]     ; a[i+1] = b[i+1] * c[i+1]
   fld  QWORD PTR [ebx+edx*8+ARR_SIZE+16]    ; b[i+2]
   fmul QWORD PTR [ecx+edx*8+ARR_SIZE+16]    ; b[i+2]*c[i+2]
   fstp QWORD PTR [eax+edx*8+ARR_SIZE+16]    ; a[i+2] = [i+2] * c[i+2]
   fld  QWORD PTR [ebx+edx*8+ARR_SIZE+24]    ; b[i+3]
   fmul QWORD PTR [ecx+edx*8+ARR_SIZE+24]    ; b[i+3] * c[i+3]
   fstp QWORD PTR [eax+edx*8+ARR_SIZE+24]    ; a[i+3] = b[i+3] * c[i+3]
   fld  QWORD PTR [ebx+edx*8+ARR_SIZE+32]    ; b[i+4]
   fmul QWORD PTR [ecx+edx*8+ARR_SIZE+32]    ; b[i+4] * c[i+4]
   fstp QWORD PTR [eax+edx*8+ARR_SIZE+32]    ; a[i+4] = b[i+4] * c[i+4]
   fld  QWORD PTR [ebx+edx*8+ARR_SIZE+40]    ; b[i+5]
   fmul QWORD PTR [ecx+edx*8+ARR_SIZE+40]    ; b[i+5] * c[i+5]
   fstp QWORD PTR [eax+edx*8+ARR_SIZE+40]    ; a[i+5] = b[i+5] * c[i+5]
   fld  QWORD PTR [ebx+edx*8+ARR_SIZE+48]    ; b[i+6]
   fmul QWORD PTR [ecx+edx*8+ARR_SIZE+48]    ; b[i+6] * c[i+6]
   fstp QWORD PTR [eax+edx*8+ARR_SIZE+48]    ; a[i+6] = b[i+6] * c[i+6]
   fld  QWORD PTR [ebx+edx*8+ARR_SIZE+56]    ; b[i+7]
   fmul QWORD PTR [ecx+edx*8+ARR_SIZE+56]    ; b[i+7] * c[i+7]
   fstp QWORD PTR [eax+edx*8+ARR_SIZE+56]    ; a[i+7] = b[i+7] * c[i+7]
   add  edx, 8   ; Compute next 8 products
   jnz  loop     ;  until none left.

END
```

The following optimization rules are applied to this example:

- Partially unroll loops to ensure that the data stride per loop iteration is equal to the length of a cache line. This avoids overlapping PREFETCH instructions and thus makes optimal use of the available number of outstanding prefetches.
- Because the array `array_a` is written rather than read, use PREFETCHW instead of PREFETCH to avoid overhead for switching cache lines to the correct state. The prefetch distance is optimized such that each loop iteration is working on three cache lines while active prefetches bring in the next cache lines.

- Reduce index arithmetic to a minimum by use of complex addressing modes and biasing of the array base addresses in order to cut down on loop overhead.

Determining Prefetch Distance

When determining how far ahead to prefetch, the basic guideline is to initiate the prefetch early enough so that the data is in the cache by the time it is needed.

To determine the optimal prefetch distance, use empirical benchmarking when possible. Prefetching four to eight cache lines ahead (256 to 512 bytes) is a good starting point. Trying to prefetch either too far ahead or too soon impairs performance.

Memory-Limited versus Processor-Limited Code

Software prefetching can help to hide the memory latency, but it cannot increase the total memory bandwidth. Many loops are limited by memory bandwidth rather than processor speed, as shown in Figure 1. In these cases, the best that software prefetching can do is to ensure that enough memory requests are “in flight” to keep the memory system busy all of the time. AMD Family 10h and 12h processors support a maximum of eight concurrent memory requests to different cache lines. Multiple requests to the same cache line count as only one towards this limit of eight.

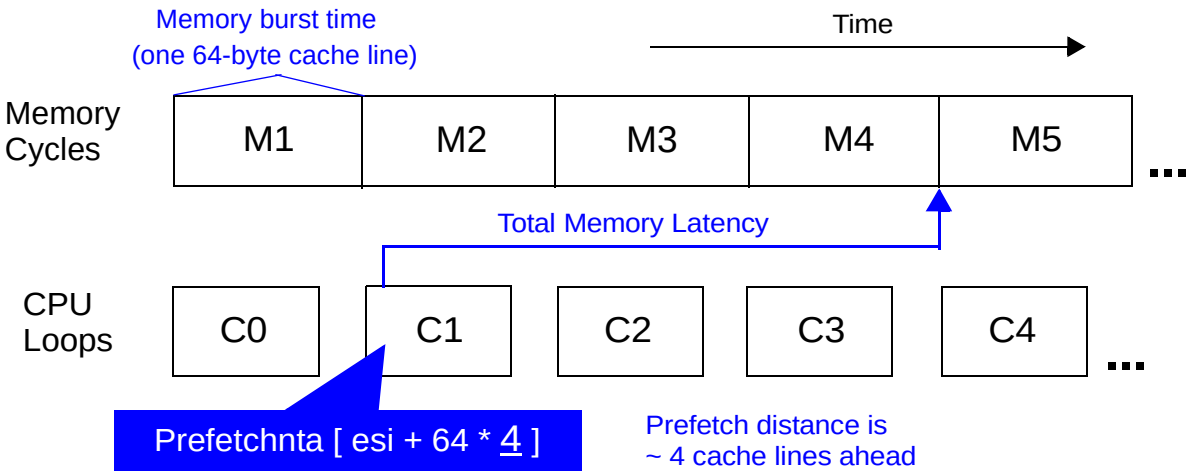


Figure 1. Memory-Limited Code

Code that performs many computations on each cache line is limited by processor speed rather than memory bandwidth, as shown in Figure 2. In this case, the goal of software prefetching is just to ensure that the memory data is available when the processor needs it. As the processor speed increases, optimal prefetch distance increases until memory bandwidth becomes the limiting factor.

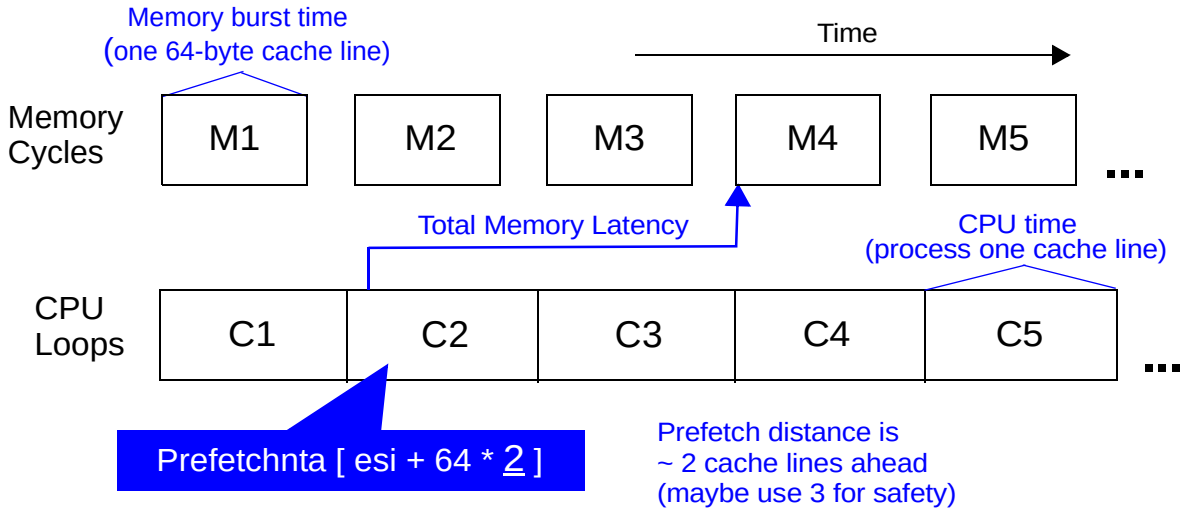


Figure 2. Processor-Limited Code

Definitions

Unit-stride access refers to a memory access pattern where consecutive memory accesses are made to consecutive array elements, in ascending or descending order. If the arrays are made of elemental types, then they imply adjacent memory locations as well. For example:

```
char j, k[MAX];
for (i = 0; i < MAX; i++) {
...
j += k[i];   // Every byte is used.
...
}
double x, y[MAX];
for (i = 0; i < MAX; i++) {
...
x += y[i];   // Every byte is used.
...
}
```

Exception to Unit Stride

The unit-stride concept works well when stepping through arrays of elementary data types. In some instances, unit stride alone may not be sufficient to determine how to use the PREFETCH instruction properly. For example, assume that there is a vertex structure of 256 bytes and the code steps through the vertices in unit stride, but using only the x, y, z, w components, each being of type `float` (for example, the first 16 bytes of each vertex). In this case, the prefetch distance obviously should be some function of the data size structure (for a properly chosen *n*):

```
prefetch [eax+n*structure_size]
...
add      eax, structure_size
```

You should experiment to find the optimal prefetch distance; there is no formula that works for all situations.

Data Stride per Loop Iteration

Assuming unit-stride access to a single array, the data stride of a loop (the *loop stride*) refers to the number of bytes accessed in the array per loop iteration. For example:

```
   fldz
add_loop:
   fadd QWORD PTR [ebx*8+base_address]
   dec  ebx
   jnz  add_loop
```

The data stride of the above loop is eight bytes. In general, for optimal use of prefetching, the data stride per iteration is the length of a cache line (64 bytes in AMD Family 10h and 12h processors). If the loop stride is smaller, unroll the loop enough to use a whole cache line of data per iteration. However, unrolling the loop may not be feasible if the original loop stride is very small (for example, only two bytes).

Prefetch at Least 64 Bytes Away from Surrounding Stores

The prefetch instructions can be affected by false dependencies on stores. If there is a store to an address that matches a request, that request (the prefetch instruction) may be blocked until the store is written to the cache. Therefore, code should prefetch data that is located at least 64 bytes away from any surrounding store's data address.

5.6 Write-Combining

Optimization

❖ Operating-system, device-driver, and BIOS programmers should take advantage of the write-combining capabilities of AMD Family 10h and 12h processors.

For details, see Appendix B, "Implementation of Write-Combining." For more information on write-combining, see "Write-Combining" in the *AMD64 Architecture Programmer's Manual, Volume 2*, order# 24593.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

In order to improve system performance, AMD Family 10h and 12h processors aggressively combine multiple memory-write cycles (of any data size) that address locations within a 64-byte cache-line-aligned write buffer.

5.7 L1 Data Cache Bank Conflicts

Optimization

Utilize pair loads that do not have a bank conflict in the L1 data cache to improve load throughput.

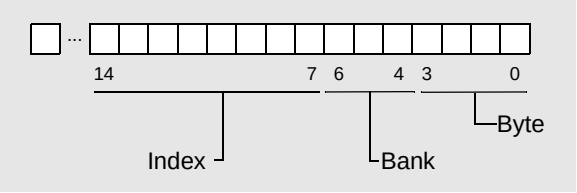
Application

This optimization applies to:

- 32-bit software
- 64-bit software

Fields Used to Address the Multibank L1 Data Cache

The L1 data cache is a multibank design consisting of eight banks total, where each bank is 16 bytes wide. To address the L1 data cache, the processor uses fields within the address as shown in the following diagram:



How to Know If a Bank Conflict Exists

The existence of a bank conflict between two neighboring loads depends on their bank and index values:

When the bank is	And the index is	Then a bank conflict
Different	Either the same or different	Does not exist
The same	The same	Does not exist
The same	Different	Exists

In other words, with common data types, consecutive array elements cannot have a bank conflict. If the array elements are 8 bytes or less, the two loads are to the same index and the same bank, and no

conflict occurs. If the array elements are 16 bytes, the loads are to the same index but different banks, so a bank conflict does not occur either.

Rationale

Loads are served by the L1 data cache in program order, but the number of loads that the processor can perform in one cycle depends on whether a bank conflict exists between the loads:

When a bank conflict	Then the number of loads the processor can perform per cycle is
Exists	1
Does not exist	2

Therefore, pairing loads that do not have a bank conflict helps maximize load throughput.

Example

Avoid code like this, where two loads without a bank conflict are separated by other instructions:

```
fld   qword ptr [eax]
fmul  qword ptr [ebx]
faddp st(3), st
fld   qword ptr [eax+16]
fmul  qword ptr [ebx+16]
faddp st(2), st
```

Instead, rearrange the two loads so they appear as a pair:

```
fld   qword ptr [eax]
fld   qword ptr [eax+16]
fmul  qword ptr [ebx+16]
faddp st(2), st
fmul  qword ptr [ebx]
faddp st(3), st
```

5.8 Placing Code and Data in the Same 64-Byte Cache Line

Optimization

❖ Avoid placing code and data together within a cache line, especially if the data becomes modified.

Application

This optimization applies to:

- 32-bit software

- 64-bit software

Rationale

Sharing code and data in the same 64-byte cache line may cause the L1 caches to thrash (unnecessarily cast out code or data) in order to maintain coherency between the separate instruction and data caches. AMD Family 10h and 12h processors have a cache-line size of 64 bytes.

For example, consider the case of a memory-indirect JMP instruction that accesses data in a jump table that resides in the same 64-byte cache line as the JMP instruction. This mixing of code and data in the same cache line degrades performance.

Do not place critical code at the border between 32-byte-aligned code segments and data segments. Code at the beginning or end of a data segment should be executed as infrequently as possible or padded.

In summary, avoid self-modifying code and storing data in code segments.

5.9 Memory and String Routines

Optimization

❖ Use the memory and string routines provided in the run-time libraries, rather than creating new custom versions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

To achieve optimal performance, it is necessary to use different memory and string manipulation algorithms to handle different block sizes and alignments. These algorithms must consider system configuration as well as the cache and memory subsystems.

The run-time libraries have optimized routines that combine several algorithms. However, if it is necessary to create fast, specific-purpose memory or string routines or to build routines that complement the run-time library, the following pseudo-code can be used as a guide to write new routines combining different algorithms:

```
if (block size is less than 8 bytes for 32 bits or less than 16 bytes for 64
bits):
  perform operations in byte, word and doubleword for 32 bits and additionally
  in quad-word for 64 bits, starting with the widest operation;

if (block size is between 8 bytes for 32 bits or 16 bytes for 64 bits and L1
    cache-line size):
  perform operations in the natural word-size in a simple loop;

if (block size is between cache-line size and smallest page-size):
  perform operations in the natural word-size in an unrolled loop,
  one cache-line size per iteration;

align the destination or a source block to the natural word-size;

if (block size is between smallest page-size and half of L1 cache-size):
  if (there is a suitable repeated string instruction)
    use repeated string instruction;
  else
    perform operations in the natural word-size in an unrolled loop, one
    cache-line size per iteration;

if (block size is between half of L1 cache-size and half of L2 cache-size)
    perform operations in the natural word-size using temporal prefetching in an
    unrolled loop, one cache-line size per iteration;

if (block size is between half of L2 cache-size and one-fourth of a core's share
     of L3 cache-size)
   perform operations in the natural word-size using non-temporal prefetching in
   an unrolled loop, one cache-line size per iteration;
```

This pseudo-code makes the following assumptions:

- Some thresholds are specified as half of a cache level because some routines have either two sources (e.g., `strcmp( )`) or a source and a destination (e.g., `memcpy( )`). A routine that has a single source or destination (e.g., `strlen( )` or `memset( )`), could use all of a cache level for its work. However, while there is usually no drawback in using all of the L1 or even L2 caches, using all of the L3 cache can hurt the performance of other processes on a system. Using only up to a core's share of the L3 cache (e.g., on a four-core processor, up to 1/4 of the L3 cache) is recommended.
- The natural word-size is a doubleword for 32 bits and a quadword for 64 bits.
- The block size thresholds between one algorithm and the other assume that the block size is unknown at the beginning. Therefore, if the block size is known beforehand to be within a certain range, experimentation may lead to different thresholds.
- Memory routines are almost completely memory bandwidth-limited; operations within loops being limited to data movement and pointers and counter maintenance. However, string routines may additionally require some computation to find the terminating null character or to ignore character case; this computation can dominate memory bandwidth. Therefore, some string routines may require many fewer algorithms than memory routines.

- Each core on a processor has access to exclusive L1 and L2 data cache and to a shared L3 cache.
- Instead of using the L2 cache-size as a threshold, particular needs and experimentation may favor using the L3 cache-size as a threshold.
- When software prefetching is used, the distance is typically eight cache lines, but experimentation may lead to a different distance.

The current generation of AMD64 processors has:

- L1 cache line-size of 64 bytes.
- Smallest page-size of 4096 bytes.
- L1 data cache size of 64 Kbytes.
- L2 cache size between 512 Kbytes and 1 Mbyte.
- L3 cache size between zero and 8 Mbytes.

See also Section 5.5, “Prefetch and Streaming Instructions” on page 81, and Section 8.3, “Repeated String Instructions” on page 128.

5.10 Stack Considerations

Optimization

Make sure the stack is suitably aligned for the local variable with the largest base type. Then, using the technique described in Section 2.16, “Sorting and Padding C and C++ Structures’ on page 31, all variables can be properly aligned with no padding.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Aligning the Stack for Local Variables

A calling convention requires a certain stack alignment on function entry. For example, the Win32 32-bit ABI arranges for 32-bit stack alignment.

If a function has no local variables with a base type larger than the guaranteed stack alignment, no further work is necessary. If the function has local variables whose base type is larger than a doubleword, insert additional code to ensure proper alignment of the stack. For example, SSE packed data requires 16-byte alignment. The following code achieves double quadword (16-byte) alignment:

```
prologue:
   push ebp
```

```
   mov  ebp, esp
   sub  esp, SIZE_OF_LOCALS   ; Size of local variables
   and  esp, -16
   ...                        ; Push registers that need to be preserved.

epilogue:                     ; Pop register that needed to be preserved.
   leave
   ret
```

For functions which have local variables that need 8-byte alignment, change the above code to use:

```
   and esp, -8
```

With this technique, function arguments can be accessed through EBP, and local variables can be accessed through ESP. Save and restore EBP between the prologue and the epilogue to keep it free for general use.

5.11 Cache Issues When Writing Instruction Bytes to Memory

Optimization

When writing data consisting of instructions for future execution to memory use streaming store (write-combining) instructions such as MOVNTDQ and MOVNTI.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

This optimization pertains to software that writes executable instructions to memory for subsequent execution, such as might be done by a just-in-time compiler. If normal store instructions are used to write the code to memory, then the cache lines will be in a modified state (either in L1 data cache or in L2). When the processor eventually tries to execute the code, it will miss in the instruction cache. Because the instruction cache cannot contain cache lines that are in a modified state, the data must be flushed to memory before it can be fetched into the instruction cache. This unnecessarily evicts possibly useful information from the caches. By using write-combining instructions, the contents of the cache is preserved with no performance penalty, and this possibly provides a performance improvement.

5.12 Interleave Loads and Stores

Optimization

When loading and storing data as in a copy routine, the organization of the sequence of loads and stores can affect performance.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

When using SSE and SSE2 instructions to perform loads and stores, it is best to interleave them in the following pattern—Load, Store, Load, Store, Load, Store, etc. This enables the processor to maximize the load/store bandwidth.

If using MMX loads and stores in 32-bit mode, the loads and stores should be arranged in the following pattern—Load, Load, Store, Store, Load, Load, Store, Store, etc.

Example

The following example illustrates a sequence of 128-bit loads and stores:

```
movdqa      xmm0,[rdx+r8*8]           ; Load
movntdq     [rcx+r8*8],xmm0           ; Store
movdqa       xmm1,[rdx+r8*8+16]       ; Load
movntdq     [rcx+r8*8+16],xmm1        ; Store
```

5.13 Using 1-Gbyte Virtual Memory Pages

Optimization

Although AMD Family 10h and 12h processor functionally supports 1-Gbyte pages for either code or data, for best performance, 1-Gbyte pages should only be used for data. They should not be used for code or for code mixed with data.

Application

This optimization applies to:

- 32-bit software

- 64-bit software

Rationale

Refer to Appendix A and you will see that there are no ITLBs for 1-Gbyte pages. Microarchitectural trade-offs were made such that 1-Gbyte pages which map to ITLBs have poor performance themselves, and in addition they cause poor performance for all other pages. Thus the use of 1-Gbyte pages for pure code or for code mixed with data is not recommended.

Chapter 6 Branch Optimizations

The optimizations in this chapter help improve branch prediction and minimize branch penalties.

This chapter covers the following topics:

Topic	Page
Branch Alignment and Density	99
Three-Byte Return-Immediate RET Instruction	100
Branches That Depend on Random Data	101
Pairing CALL and RETURN	103
Nonzero Code-Segment Base Values	104
Replacing Branches	104
Avoiding the LOOP Instruction	106
Far Control-Transfer Instructions	106
Branches Not-Taken Preferable to Branches Taken	107

6.1 Branch Alignment and Density

Optimization

When possible, align branch targets to a 32-byte boundary and limit the number of branches in a 16-byte boundary to three.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

AMD Family 10h and 12h processors have the capability to cache the branch-prediction history for a maximum of three near branches (CALL, JMP, conditional branches, or returns) per 16-byte fetch window. A branch instruction that crosses a 16-byte boundary is counted in the second 16-byte window. Due to architectural restrictions, a branch that is split across a 16-byte boundary cannot dispatch with any other instructions when it is predicted to be taken. Perform this alignment by rearranging code; it is not beneficial to align branches using padding sequences.

The branch prediction hardware can only support up to three near branches per aligned 16 byte window. Coding more than three branches in the same 16-byte code window may lead to conflicts in

prediction storage. To avoid conflicts in branch prediction storage, space out branches in such a way that three or fewer exist in a given 16-byte code window, with one of the three being preferably an unconditional branch if at all possible. For absolute optimal performance, try to limit branches to one per 16-byte code window. If there is a jump table that contains many frequently executed branches, pad the table entries to 8 bytes each to assure that there are never more than three branches per 16-byte block of code. Note that a branch is assigned to a region based on its end byte and not its start byte.

Only branches that have been taken at least once are entered into the branch prediction, and therefore only those branches count toward the three-branch limit.

By aligning branch targets to 32-byte boundaries, the number of instructions in a fetch-window to be processed by the following stages is maximized.

6.2 Three-Byte Return-Immediate RET Instruction

Optimization

❖ Use of a three-byte return-immediate can improve performance. The single-byte near-return (opcode C3h) of the RET instruction should be used carefully. Specifically, avoid the following two situations:

- Any kind of branch (either conditional or unconditional) that has the single-byte near-return RET instruction as its target. See “Examples” on page 101.
- A conditional branch that occurs in the code directly before the single-byte near-return RET instruction. See “Examples” on page 101

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The processor is sometimes unable to apply a branch prediction to the single-byte near-return form (opcode C3h) of the RET instruction.

The easiest way to assure the utilization of the branch prediction mechanism is to use a three-byte RET *imm16* instruction with an *imm16* value of 0, which produces the functional equivalent of the single-byte near-return RET instruction, but is not affected by the prediction limitations outlined above. To use a three-byte RET *imm16* with an *imm16* value of 0, define a text macro named RETIMM0 and use it instead of the RET instruction to force the intended object code.

```
RETIMM0 TEXTEQU <DB 0C2h, 0, 0>
```

Examples

Avoid branches in which the target of the branch is a single-byte near-return:

```
   jmp label   ; Jump to a single-byte near-return RET instruction.
   ...
label:
   ret         ; RET is potentially mispredicted.
```

Avoid branches that immediately precede a single-byte near-return:

```
jz  label   ; Conditional branch is not taken.
ret         ; RET is a fall-through instruction,
            ;  potentially mispredicted.
```

If possible, move an existing instruction, such as a POP instruction that is part of the function epilogue, so that it is inserted between the branch and the RET instruction:

```
jz  label
pop ebp   ; Pad with at least one non-branch instruction.
ret
```

If no existing instruction is available for this purpose, then insert a NOP instruction to provide the necessary padding or, better still, use the recommended three-byte version of RET *imm16*.

6.3 Branches That Depend on Random Data

Optimization

❖ Avoid conditional branches that depend on random data, as these branches are difficult to predict.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Suppose a piece of code receives a random stream of characters “A” through “Z” and branches if the character is before “M” in the collating sequence. Data-dependent branches acting upon basically random data cause the branch-prediction logic to mispredict the branch about 50% of the time.

If possible, design branch-free alternative code sequences that result in shorter average execution time. This technique is especially important if the branch body is small.

Examples

The following examples illustrate this concept using the CMOV*xx* instruction.

Signed Integer ABS Function (x = labs(x))

```
mov   ecx, [x]   ; Load value.
mov   ebx, ecx   ; Save value.
neg   ecx        ; Negate value.
cmovs ecx, ebx   ; If negated value is negative, select value.
mov   [x], ecx   ; Save labs result.
```

Unsigned Integer min Function (z = x < y ? x : y)

```
mov    eax, [x]   ; Load x value.
mov    ebx, [y]   ; Load y value.
cmp    eax, ebx   ; EBX <= EAX ? CF = 0 : CF = 1
cmovnc eax, ebx   ; EAX = (EBX <= EAX) ? EBX : EAX
mov    [z], eax   ; Save min(X,Y).
```

Conditional Write

```
// C code:

int a, b, i, dummy, c[BUFSIZE];

if (a < b) {
   c[i++] = a;
}

;--------------
; Assembly code:

lea esi, [dummy]    ; &dummy
xor ecx, ecx        ; i = 0
...
lea    edi, [c+ecx*4]   ; &c[i]
lea    edx, [ecx+1]     ; i++
cmp    eax, ebx         ; a < b ?
cmovge edi, esi         ; ptr = (a >= b) ? &dummy : &c[i]
cmovl  ecx, edx         ; a < b ? i : i + 1
mov    [edi], eax       ; *ptr = a
```

6.4 Pairing CALL and RETURN

Optimization

For each CALL to a subroutine, use a RET instruction to return to the caller.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

As explained in “Branch-Prediction Table” on page 229, the Return Address Stack (RAS) can predict a limited number of branches by the RET instruction. CALL instructions push the next rIP on the return address stack. The corresponding RET instruction uses this address for its target prediction. If the RAS overflows, then the oldest return address is lost and the corresponding RET will likely be mispredicted, considerably lengthening the latency of the RET instruction.

When a CALL instruction is not paired with a RET instruction, the RAS can get out of sync, lengthening the latency of other RET instructions whose return addresses remain in the RAS. However, there is an important special case, shown in the following example, commonly used to get the value in the EIP register into a general-purpose register in 32-bit software:

```
CALL 0h
POP  EAX   ; EAX contains the value of EIP
```

When the CALL instruction is used with a displacement of zero, it is recognized and treated specially; the RAS remains consistent even if there is not a corresponding RET instruction.

To get the value in the RIP register into a general-purpose register in 64-bit software, you can use RIP-relative addressing, as in the following example:

```
LEA  RAX, [RIP+0] ; RAX contains the value of RIP.
```

6.5 Nonzero Code-Segment Base Values

Optimization

In 32-bit threads, avoid using a nonzero code-segment (CS) base value. (In 64-bit mode, segmentation is disabled and the segment base value is ignored and treated as zero.)

Application

This optimization applies to:

- 32-bit software

Rationale

A nonzero CS base value causes an additional two cycles of branch-misprediction penalty when compared with a CS base value of zero.

6.6 Replacing Branches

Optimization

Use muxing constructs to simulate conditional moves in SSE or MMX code.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Branches can negatively impact the performance of code. In SSE or MMX code, if the body of the branch is small, you can achieve higher performance instead computing both paths of the branch and using muxing constructs to construct the result. This simulates predicated execution or conditional moves. There are many SSE and SSE2 instructions that can be useful for accomplishing this. The principal instructions are as follows: ANDPS, ANDPD, ANDNPS, ANDNPD, CMPPS, CMPSS, CMPPD, CMPSD, MINPS, MINSS, MINPD, MINSD, MAXPS, MAXSS, MAXPD, MAXSD, ORPS, ORPD, PAND, PANDN, PCMPEQB, PCMPEQD, PCMPEQW, PCMPGTB, PCMPGTD, PCMPGTW, PMAXSW, PMAXUB, PMINSW, PMINUB, POR, PXOR, XORPS, and XORPD.

When using MMX registers, the following instructions may be useful for eliminating branches: PCMPGTB, PCMPGTD, PCMPGTW, PAND, PANDN, POR, and PXOR.

Muxing Constructs

The most important construct to use in avoiding branches in SIMD code is a two-way muxing construct that is equivalent to the ternary operator (`?:`) in C and C++.

Examples

SSE Solution (Preferred)

```
; r = (x < y) ? a : b
;
; In:  XMM0 = a
;      XMM1 = b
;      XMM2 = x
;      XMM3 = y
; Out: XMM0 = r

cmpps  xmm2, xmm3, 1   ; x < y ? 0xffffffff : 0
andps  xmm0, xmm2      ; x < y ? a : 0
andnps xmm2, xmm1      ; x < y ? 0 : b
orps   xmm0, xmm2      ; x < y ? a : b
```

MMX™ Solution (Preferred)

```
; r = (x < y) ? a : b
;
; In: MM0 = a
;     MM1 = b
;     MM2 = x
;     MM3 = y
; Out: MM0 = r

pcmpgtd mm3, mm2 ; y > x ? 0xffffffff : 0
pand mm0, mm3 ; y > x ? a: 0
pandn mm3, mm1 ; y > x > 0 : b
por mm0, mm3 ; r = y > x ? a : b
```

Avoid the following muxing construct. This example reverses the order of the PAND and PANDN instructions. Because the use of PANDN destroys the mask created by PCMPGTD, the mask must be saved, requiring the use of an additional register. This adds an instruction, lengthens the dependency chain, and increases register pressure.

MMX™ Solution (Avoid)

```
; r = (x < y) ? a : b
;
; In:  MM0 = a
;      MM1 = b
;      MM2 = x
;      MM3 = y
; Out: MM0 = r
```

```
pcmpgtd   mm3, mm2   ; y > x ? 0xffffffff : 0
movq      mm4, mm3   ; Duplicate mask
pandn     mm3, mm1   ; y > x ? 0: b
pand      mm0, mm4   ; y > x ? a : 0
```

6.7 Avoiding the LOOP Instruction

Optimization

Avoid using the LOOP instruction.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The LOOP instruction has a latency of 7 cycles in 32-bit protected mode and 8 cycles in 64-bit protected mode.

Example

Avoid code like this, which uses the LOOP instruction:

```
label:
   ...
   loop label         ;Latency is 7/8 cycles, depending upon whether we
                      ; are in 32-bit or 64-bit protected mode.
```

Instead, replace the loop instruction with a DEC and a JNZ:

```
label:
   ...
   dec rcx            ;Latency of 1 cycle for register operand form of DEC.
   jnz label          ;Latency of 1 cycle.
```

6.8 Far Control-Transfer Instructions

Optimization

Use far control-transfer instructions only when necessary. (Far control-transfer instructions include the far forms of JMP, CALL, and RET, as well as the INT, INTO, and IRET instructions.)

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The processor's branch-prediction unit does not predict far branches.

6.9 Branches Not-Taken Preferable to Branches Taken

Optimization

Whenever possible, use branches that are biased toward being not-taken over branches that are biased toward being taken.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Correctly-predicted taken branches have at least one prediction-based bubble while not-taken branches do not. In addition, taken branches consume more branch prediction resources.

Chapter 7 Scheduling Optimizations

The optimizations discussed in this chapter help improve scheduling in the processor.

This chapter covers the following topics:

Topic	Page
Instruction Scheduling by Latency	109
Loop Unrolling	110
Inline Functions	113
Address-Generation Interlocks	115
MOVZX and MOVSX	116
Pointer Arithmetic in Loops	116
Pushing Memory Data Directly onto the Stack	118

7.1 Instruction Scheduling by Latency

Optimization

In general, select instructions with shorter latencies that are DirectPath—not VectorPath—instructions. For a list of instruction latencies and classifications, see Appendix C, “Instruction Latencies.”

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

AMD Family 10h and 12h processors can execute up to three AMD64 instructions per cycle, with each instruction possibly having a different latency. AMD Family 10h and 12h processors have flexible scheduling, but for absolute maximum performance, schedule instructions according to their latencies and data dependencies. The goal is to reduce the overall length of dependency chains.

7.2 Loop Unrolling

Optimization

Use loop unrolling where appropriate to increase instruction-level parallelism:

If all of these conditions are true	Then use
• The loop is in a frequently executed piece of code. • The number of loop iterations is known at compile time. • The loop body includes fewer than 10 instructions.	Complete loop unrolling
• Spare registers are available (for example, when operating in 64-bit mode, where additional registers are available). • The loop body is small, so that loop overhead is significant. • The number of loop iterations is likely greater than 10.	Partial loop unrolling

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Loop Unrolling

Loop unrolling is a technique that duplicates the body of a loop one or more times in order to increase the number of instructions relative to the branch and allow operations from different loop iterations to execute in parallel.

There are two types of loop unrolling:

- Complete loop unrolling
- Partial loop unrolling

Complete Loop Unrolling

Complete loop unrolling eliminates the loop overhead completely by replacing the loop with copies of the loop body.

Because complete loop unrolling removes the loop counter, it also reduces register pressure. However, completely unrolling very large loops can result in the inefficient use of the L1 instruction cache.

Example—Complete Loop Unrolling

In the following C code, the number of loop iterations is known at compile time and the loop body is less than 100 instructions:

```
#define ARRAY_LENGTH 3

int sum, i, a[ARRAY_LENGTH];

...
sum = 0;
for (i = 0; i < ARRAY_LENGTH; i++) {
   sum = sum + a[i];
}
```

To completely unroll an *n*-iteration loop, remove the loop control and replicate the loop body *n* times:

```
sum = 0;
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
```

Partial Loop Unrolling

Partial loop unrolling reduces the loop overhead by duplicating the loop body several times, changing the increment in the loop, and adding cleanup code to execute any leftover iterations of the loop. The number of times the loop body is duplicated is known as the *unroll factor.*

However, partial loop unrolling may increase register pressure.

Example—Partial Loop Unrolling

In the following C code, each element of one array is added to the corresponding element of another array:

```
double a[MAX_LENGTH], b[MAX_LENGTH];

for (i = 0; i < MAX_LENGTH; i++) {
   a[i] = a[i] + b[i];
}
```

Without loop unrolling, this is the equivalent assembly-language code:

```
   mov ecx, MAX_LENGTH   ; Initialize counter.
   mov eax, OFFSET a     ; Load address of array a into EAX.
   mov ebx, OFFSET b     ; Load address of array b into EBX.

add_loop:
   movsd  xmm0, QWORD PTR [eax] ; Load double pointed to by EAX
   addsd  xmm0, QWORD PTR [ebx] ; Add double pointed to by EBX
   movsd  QWORD PTR [eax], xmm0 ; Store double result.
   add  eax, 8            ; Point to next element of array a.
   add  ebx, 8            ; Point to next element of array b.
```

```
   dec  ecx               ; Decrement counter.
   jnz  add_loop          ; If elements remain, then jump.
```

The rolled loop consists of seven instructions. AMD Family 10h and 12h processors can decode and retire as many as three instructions per cycle, so it cannot execute faster than three iterations in seven cycles (3/7 of a floating-point add per cycle). However, the pipelined floating-point adder allows one add every cycle.

$$\frac{3\ \text{instructions}}{\text{cycle}} \times \frac{\text{iteration}}{7\ \text{instructions}} \times \frac{1\ \text{FADD}}{\text{iteration}} = \frac{3\ \text{FADDs}}{7\ \text{cycles}} = 0.429\text{FADDs/cycle}$$

After partial loop unrolling using an unroll factor of two, the new code creates a potential end case that must be handled outside the loop:

```
   mov ecx, MAX_LENGTH   ; Initialize counter.
   mov eax, OFFSET a     ; Load address of array a into EAX.
   mov ebx, OFFSET b     ; Load address of array b into EBX.

   shr  ecx, 1            ; Divide counter by 2 (the unroll factor).
   jnc  add_loop          ; If original counter was even, then jump.
   ; Handle the end case.
   movsd  xmm0, QWORD PTR [eax] ; Load double pointed to by EAX
   addsd   xmm0, QWORD PTR [ebx] ; Add double pointed to by EBX
   movsd  QWORD PTR [eax], xmm0 ; Store double result.
   add  eax, 8            ; Point to next element of array a.
   add  ebx, 8            ; Point to next element of array b.

add_loop:
   movsd  xmm0, QWORD PTR [eax]  ; Load double pointed to by EAX
   addsd   xmm0, QWORD PTR [ebx] ; Add double pointed to by EBX
   movsd  QWORD PTR [eax], xmm0 ; Store double result.
   movsd  xmm0, QWORD PTR [eax+8]  ; repeat for next double
   addsd   xmm0, QWORD PTR [ebx+8] ; Add double pointed to by EBX
   movsd  QWORD PTR [eax], xmm0 ; Store double result.
   add  eax, 16             ; Point to next element of array a.
   add  ebx, 16             ; Point to next element of array b.
   dec  ecx                 ; Decrement counter.
   jnz  add_loop            ; If elements remain, then jump.
```

The unrolled loop consists of 10 instructions. Based on the decode/retire bandwidth of three instructions per cycle, this loop goes no faster than three iterations in 10 cycles (which is equivalent to 6/10 of a floating-point add per cycle because there are two additions per iteration), or 1.4 times as fast as the original loop.

$$\frac{3\ \text{instructions}}{\text{cycle}} \times \frac{\text{iteration}}{10\ \text{instructions}} \times \frac{2\ \text{FADDs}}{\text{iteration}} = \frac{6\ \text{FADDs}}{10\ \text{cycles}} = 0.600\ \text{FADDs § cycle}$$

Deriving the Loop Control for Partially Unrolled Loops

A frequently used loop construct is a counting loop. In a typical case, the loop count starts at some lower bound (`low`), increases by some fixed, positive increment (`inc`) for each iteration of the loop, and may not exceed some upper bound (`high`):

```
for (k = low; k <= high; k += inc) {
   x[k] = ...
}
```

The following code shows how to partially unroll such a loop by an unroll factor (`factor`) and how to derive the loop control for the partially unrolled version of the loop:

```
for (k = low; k <= (high - (factor - 1) * inc); k += factor * inc) {
   // Begin the series of unrolled statements.
   x[k + 0 * inc] = ...
   // Continue the series if the unrolling factor is greater than 2.
   x[k + 1 * inc] = ...
   x[k + 2 * inc] = ...
   ...
   // End the series.
   x[k + (factor - 1) * inc] = ...
}

// Handle the end cases.
for (k = k; k <= high; k += inc) {
   x[k] = ...
}
```

Related Information

For information on loop unrolling at the C-source level, see “Unrolling Small Loops” on page 17.

7.3 Inline Functions

Optimization

Use function inlining when:

- A function is called from just one site in the code. (For the C language, determination of this characteristic is made easier if functions are explicitly declared `static` unless they require external linkage.)
- A function—once inlined—contains fewer than 25 machine instructions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

There are advantages and disadvantages to function inlining. On the one hand, function inlining eliminates function-call overhead and allows better register allocation and instruction scheduling at the site of the function call. The disadvantage of function inlining is decreased code reference locality, which can increase execution time due to instruction cache misses.

For functions that create fewer than 25 machine instructions once inlined, it is likely that the function-call overhead is close to, or more than, the time spent executing the function body. In these cases, function inlining is recommended.

Function-call overhead on the AMD Family 10h and 12h processors can be low because calls and returns are executed very quickly due to the use of prediction mechanisms. However, there is still overhead due to passing function arguments through memory, which creates store-to-load-forwarding dependencies. (In 64-bit mode, this overhead is typically avoided by passing more arguments in registers, as specified in the *AMD64 Application Binary Interface* [ABI] for the operating system.)

For longer functions, inlining yields diminishing returns. A function that results in the insertion of more than 500 machine instructions at the call site should probably not be inlined. Some larger functions might consist of multiple, relatively short paths. The execution time of the body of such a function may be relatively short compared to the function overhead, in which case inlining can improve performance. Profiling information is the best guide in determining whether to inline such large functions.

Additional Recommendations for Compiler Writers

In general, function inlining works best if the compiler utilizes feedback from a profiler to identify the function calls most frequently executed. If such data is not available, a reasonable approach is to concentrate on function calls inside loops. Do not consider as candidates for inlining any functions that are directly recursive. However, if they are end-recursive, the compiler should convert them to an iterative equivalent to avoid potential overflow of the processor's return-prediction mechanism (return stack) during deep recursion. For best results, a compiler should support function inlining across multiple source files. In addition, a compiler should provide intrinsic functions for commonly used library routines, such as `sin`, `strcmp`, or `memcpy`.

7.4 Address-Generation Interlocks

Optimization

Avoid address-generation interlocks by scheduling loads and stores whose addresses can be calculated quickly ahead of loads and stores that require the resolution of a long dependency chain in order to generate their addresses.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Address-Generation Interlocks

An *address-generation interlock* is a condition in which newer loads and stores whose addresses have already been calculated by the processor are blocked by older loads and stores whose addresses have not yet been calculated.

Rationale

The processor schedules instructions that access the data cache (loads and stores) in program order. By carefully choosing the order of loads and stores, you can avoid address-generation interlocks.

Example

Avoid code that places a load whose address takes longer to calculate before a load whose address can be determined more quickly:

```
mov eax, DWORD PTR [10h]        ; Instruction 1, generates address for 2.
mov DWORD PTR [eax],ecx         ; Instruction 2, depends on 1.
mov edx, DWORD PTR [24h]        ; Instruction 3, independent.
```

When scheduled in this order, Instruction 3 will be stalled while waiting for Instruction 1 to generate the address required by Instruction 2.

The following instruction schedule is preferable.

```
mov eax, DWORD PTR [10h]        ; Instruction 1, generates address for 2.
mov edx, DWORD PTR [24h]        ; Instruction 3, independent.
mov DWORD PTR [eax],ecx         ; Instruction 2, depends on 1.
```

This scheduling allows Instruction 3 to proceed while Instruction 2 waits for his address to be ready. In fact, up to five independent loads can profitably be scheduled between 1 and 2.

Where possible, reorder instructions so that loads with independent addresses come between pairs of dependent loads.

7.5 MOVZX and MOVSX

Optimization

Use the MOVZX and MOVSX instructions to zero-extend or sign-extend, respectively, an operand to a larger size.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Typical code for zero extension that replaces MOVZX uses more decode and execution resources than MOVZX. It also has higher latency due to the superset dependency between the XOR and the MOV, which requires a merge operation.

Example

When zero-extending an operand (in this case, a byte), avoid code such as the following:

```
xor rax, rax
mov al, mem
```

Instead, use the MOVZX instruction:

```
movzx rax, BYTE PTR mem
```

7.6 Pointer Arithmetic in Loops

Optimization

Minimize pointer arithmetic in loops, especially if the loop bodies are small. Take advantage of scaled-index addressing modes to utilize the loop counter as an index into memory arrays.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

In small loops, pointer arithmetic causes significant overhead. Using scaled-index addressing modes has no negative impact on execution speed, but the reduced number of instructions preserves decode bandwidth.

Example

Consider the following C code, which adds the elements of two arrays and stores them in a third array:

```
int a[MAXSIZE], b[MAXSIZE], c[MAXSIZE], i;

for (i = 0; i < MAXSIZE; i++) {
   c[i] = a[i] + b[i];
}
```

Avoid an assembly-language equivalent like this, which uses base and displacement components (for example, `[esi+a]`) to compute array-element addresses, requiring additional pointer arithmetic to increment the offsets into the forward-traversed arrays:

```
   mov ecx, MAXSIZE   ; Initialize loop counter.
   xor esi, esi       ; Initialize offset into array a.
   xor edi, edi       ; Initialize offset into array b.
   xor ebx, ebx       ; Initialize offset into array c.

add_loop:
   mov eax, [esi+a]   ; Get element from a.
   mov edx, [edi+b]   ; Get element from b.
   add eax, edx       ; a[i] + b[i]
   mov [ebx+c], eax   ; Write result to c.
   add esi, 4         ; Increment offset into a.
   add edi, 4         ; Increment offset into b.
   add ebx, 4         ; Increment offset into c.
   dec ecx            ; Decrement loop count
   jnz add_loop       ;  until loop count is 0.
```

Instead, traverse the arrays in a downward direction (from higher to lower addresses), in order to take advantage of scaled-index addressing (for example, `[ecx*4+a]`), which minimizes pointer arithmetic within the loop:

```
   mov ecx, MAXSIZE - 1   ; Initialize index.

add_loop:
   mov eax, [ecx*4+a]   ; Get element from a.
   mov edx, [ecx*4+b]   ; Get element from b.
   add eax, edx         ; a[i] + b[i]
   mov [ecx*4+c], eax   ; Write result to c.
```

```
   dec ecx              ; Decrement index
   jns add_loop         ;  until index is negative.
```

A change in the direction of traversal is possible only if each loop iteration is completely independent of the others. If you cannot change the direction of traversal for a given array, it is still possible to minimize pointer arithmetic by using as a base address a displacement that points to the byte past the end of the array, and using an index that starts with a negative value and reaches zero when the loop expires:

```
   mov ecx, (-MAXSIZE)   ; Initialize index.

add_loop:
   mov eax, [ecx*4+a+MAXSIZE*4]   ; Get element from a.
   mov edx, [ecx*4+b+MAXSIZE*4]   ; Get element from b.
   add eax, edx                   ; a[i] + b[i]
   mov [ecx*4+c+MAXSIZE*4], eax   ; Write result to c.
   inc ecx                        ; Increment index
   jnz add_loop                   ;  until index is 0.
```

If the base addresses of the arrays are held in registers (for example, when the base addresses are passed as the arguments of a function), biasing the base addresses requires additional instructions to perform the biasing at run time, and a small amount of additional overhead is incurred.

7.7 Pushing Memory Data Directly onto the Stack

Optimization

Push memory data directly onto the stack instead of loading it into a register first.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Pushing memory data directly onto the stack reduces register pressure and eliminates data dependencies.

Example

Avoid code that first loads the memory data into a register and then pushes it onto the stack:

```
mov  rax, mem
push rax
```

Instead, push the memory data directly onto the stack:

```
push mem
```

Chapter 8 Integer Optimizations

The optimizations in this chapter help improve integer performance.

This chapter covers the following topics:

Topic	Page
Replacing Division with Multiplication	121
Alternative Code for Multiplying by a Constant	125
Repeated String Instructions	128
Using XOR to Clear Integer Registers	129
Efficient 64-Bit Integer Arithmetic in 32-Bit Mode	130
Derivation of Algorithm, Multiplier, and Shift Factor for Integer Division by Constants	137
Optimizing Integer Division	143
Efficient Implementation of Population Count and Leading-Zero Count	144

8.1 Replacing Division with Multiplication

Optimization

Replace integer division by constants with multiplication by the reciprocal.

Rationale

For AMD Family 12h it is recommended to use the division instructions DIV or IDIV (refer to Appendix C to review the latency of DIV/ IDIV instructions). DIV and IDIV instructions are now fastpath instructions. The divider is not pipelined and can only execute one div instruction at any given point of time. The latency of unit to produce the quotient is 8+ ceil(num quotient bits/2). The remainder is available 2 cycles later.

The remaining information in this section 8.1 applies to AMD Family 10h processors only.

AMD Family 10h processors have very fast integer multiplication instructions (IMUL, MUL) whereas the integer division instructions (IDIV and DIV) are vector instructions having a variable latency that depends on the number of bits in the divisor. (For exact latencies, see “Optimizing Integer Division” on page 143 and Appendix C, “Instruction Latencies.”)

For this reason division by a constant should be replaced by multiplication by the reciprocal of the constant. The exact code to use for multiplication by the reciprocal of the constant can be found either in the examples later in this section or by using the utilities in “Derivation of Algorithm, Multiplier, and Shift Factor for Integer Division by Constants” on page 137.

Multiplication by Reciprocal (Division) Utility

The code for the utilities is shown in "Derivation of Algorithm, Multiplier, and Shift Factor for Integer Division by Constants" on page 137. The utilities provided in this document are for reference only and are not supported by AMD.

Signed Division Utility

The sdiv.exe utility finds the fastest code for *signed* division by a constant. The utility displays the code after the user enters a signed constant divisor. To redirect the code to a file, type the following command:

```
sdiv > example.out
```

Unsigned Division Utility

The udiv.exe utility finds the fastest code for *unsigned* division by a constant. The utility displays the code after the user enters an unsigned constant divisor. To redirect the code to a file, type the following command:

```
udiv > example.out
```

Unsigned Division by Multiplication of Constant

Algorithm: Divisors $1 \leq d < 2^{31}$, Odd d

The following code shows an unsigned division using a constant value multiplier.

```
; a = algorithm
; m = multiplier
; s = shift factor

; a == 0
mov eax, m
mul dividend
shr edx, s   ; EDX = quotient

; a == 1
mov eax, m
mul dividend
add eax, m
adc edx, 0
shr edx, s   ; EDX = quotient
```

Code for determining the algorithm (*a*), multiplier (*m*), and shift factor (*s*) from the divisor (*d*) is found in the section "Derivation of Algorithm, Multiplier, and Shift Factor for Integer Division by Constants" on page 137.

Algorithm: Divisors $2^{31} \leq d < 2^{32}$

For divisors $2^{31} \leq d < 2^{32}$, the possible quotient values are either 0 or 1. For this reason, it is easy to establish the quotient by simple comparison of the dividend and divisor. When the dividend needs to be preserved, consider using code like the following:

```
; In:  EAX = dividend
; Out: EDX = quotient

xor edx, edx   ; 0
cmp eax, d     ; CF = (dividend < divisor) ? 1 : 0
sbb edx, -1    ; quotient = 0 + 1 - CF = (dividend < divisor) ? 0 : 1
```

When it is not necessary to preserve the dividend, the division can be accomplished without the use of an additional register, thus reducing register pressure, as shown in the following example:

```
; In:  EAX = dividend
; Out: EDX = quotient

cmp edx, d    ; CF = (dividend < divisor) ? 1 : 0
mov eax, 0    ; 0
sbb eax, -1   ; quotient = 0 + 1 - CF = (dividend < divisor) ? 0 : 1
```

Simpler Code for Restricted Dividend

Integer division by a constant can be accelerated by limiting the range of the dividend, which removes a shift associated with most divisors. For example, for a divide-by-10 operation, use the following code, if the dividend is less than 4000_0005h:

```
mov eax, dividend
mov edx, 01999999Ah
mul edx
mov quotient, edx
```

Signed Division by Multiplication of Constant

Algorithm: Divisors $2 \leq d < 2^{31}$

The following algorithms work if the divisor is positive. If the divisor is negative, use ABS(d) instead of d, and append a NEG edx instruction to the code. These changes make use of the fact that $n/–d = –(n/d)$.

```
; a is the algorithm to select between two sets of code
;      sequences depending on the calculation of multiplier.
; m is the multiplier, the constant used with the multiply instruction.
; s is the amount of right shifting to accomplish the division after the
;      multiplication of a constant.

; a == 0
mov  eax, m
imul dividend
mov  eax, dividend
```

```
shr  eax, 31
sar  edx, s
add  edx, eax       ; Quotient in EDX

; a == 1
mov  eax, m
imul dividend
mov  eax, dividend
add  edx, eax
shr  eax, 31
sar  edx, s
add  edx, eax   ; Quotient in EDX
```

Code for determining the algorithm (*a*), multiplier (*m*), and shift factor (*s*) is shown in "Derivation of Algorithm, Multiplier, and Shift Factor for Integer Division by Constants" on page 137.

Signed Division by 2

```
; In:  EAX = dividend
; Out: EAX = quotient

cmp eax, 80000000h   ; CF = 1 if dividend >= 0.
sbb eax, -1          ; Increment dividend if it is < 0.
sar eax, 1           ; Perform right shift.
```

Signed Division by 2^n

```
; In:  EAX = dividend
; Out: EAX = quotient

cdq                ; Sign extend into EDX.
and edx, (2^n - 1)   ; Mask correction (use divisor - 1)
add eax, edx         ; Apply correction if necessary.
sar eax, (n)         ; Perform right shift by log2(divisor).
```

Signed Division by –2

```
; In:  EAX = dividend
; Out: EAX = quotient

cmp eax, 80000000h   ; CF = 1 if dividend >= 0.
sbb eax, -1          ; Increment dividend if it is < 0.
sar eax, 1           ; Perform right shift.
neg eax              ; Use (x / -2) == -(x / 2).
```

Signed Division by $-(2^n)$

```
; In:  EAX = dividend
; Out: EAX = quotient

cdq                ; Sign extend into EDX.
and edx, (2^n - 1)   ; Mask correction (-divisor - 1).
add eax, edx         ; Apply correction if necessary.
sar eax, (n)         ; Right shift by log2(-divisor).
neg eax              ; Use (x / -(2^n)) == (-(x / 2^n)).
```

Remainder of Signed Division by 2 or –2

```
; In:  EAX = dividend
; Out: EAX = remainder

cdq          ; Sign extend into EDX.
and eax, 1    ; Compute remainder.
xor eax, edx   ; Negate remainder if
sub eax, edx   ;  dividend was < 0.
```

Remainder of Signed Division by 2^n or $–(2^n)$

```
; In:  EAX = dividend
; Out: EAX = remainder

cdq               ; Sign extend into EDX.
and edx, (2^n - 1)   ; Mask correction (abs(divisor) - 1)
add eax, edx         ; Apply pre-correction.
and eax, (2^n - 1)   ; Mask out remainder (abs(divisor) - 1)
sub eax, edx         ; Apply pre-correction if necessary.
```

8.2 Alternative Code for Multiplying by a Constant

Optimization

Devise instruction sequences with lower latency to accomplish multiplication by certain constant multipliers.

Rationale

A 32-bit integer multiplied by a constant has a latency of 3 cycles; a 64-bit integer multiplied by a constant has a latency of 4 cycles. For certain constant multipliers, instruction sequences can be devised that accomplish the multiplication with lower latency. Because AMD Family 10h and 12h processors contain only one integer multiplier but three integer execution units, the replacement code can provide better throughput as well.

Most replacement sequences require the use of an additional temporary register, thus increasing register pressure. If register pressure in a piece of code that performs integer multiplication with a constant is already high, it could be better for the overall performance of that code to use the IMUL instruction instead of the replacement code. Similarly, replacement sequences with low latency but containing many instructions may negatively influence decode bandwidth as compared to the IMUL instruction. In general, replacement sequences containing more than four instructions are not recommended.

The following code samples are designed for the original source to receive the final result. Other sequences are possible if the result is in a different register. Sequences that do not require a temporary register are favored over those requiring a temporary register, even if the latency is higher. To keep code size small, arithmetic-logic-unit operations are preferred over shifts. Similarly, both arithmetic-logic-unit operations and shifts are favored over the LEA instruction.

There are improvements in the AMD Family 10h and 12h processors' multiplier over that of previous x86 processors. For this reason, when doing 32-bit multiplication, only use the alternative sequence if the alternative sequence has a latency that is less than or equal to 2 cycles. For 64-bit multiplication, only use the alternative sequence if the alternative sequence has a latency that is less than or equal to 3 cycles.

Examples

```
by 2:   add reg1, reg1          ; 1 cycle

by 3:   lea reg1, [reg1+reg1*2]   ; 2 cycles

by 4:   shl reg1, 2             ; 1 cycle

by 5:   lea reg1, [reg1+reg1*4]   ; 2 cycles

by 6:   lea reg1, [reg1+reg1*2]   ; 3 cycles
        add reg1, reg1

by 7:   mov reg2, reg1          ; 2 cycles
        shl reg1, 3
        sub reg1, reg2

by 8:   shl reg1, 3             ; 1 cycle

by 9:   lea reg1, [reg1+reg1*8]   ; 2 cycles

by 10:  lea reg1, [reg1+reg1*4]   ; 3 cycles
        add reg1, reg1

by 11:  lea reg2, [reg1+reg1*8]   ; 3 cycles
        add reg1, reg1
        add reg1, reg2

by 12:  lea reg1, [reg1+reg1*2]   ; 3 cycles
        shl reg1, 2

by 13:  lea reg2, [reg1+reg1*2]   ; 3 cycles
        shl reg1, 4
        sub reg1, reg2

by 14:  lea reg2, [reg1+reg1]     ; 3 cycles
        shl reg1, 4
        sub reg1, reg2

by 15:  mov reg2, reg1          ; 3 cycles
        shl reg1, 4
        sub reg1, reg2

by 16:  shl reg1, 4             ; 1 cycle

by 17:  mov reg2, reg1          ; 2 cycles
        shl reg1, 4
        add reg1, reg2
```

```
by 18:  lea reg1, [reg1+reg1*8]   ; 3 cycles
        add reg1, reg1

by 19:  lea reg2, [reg1+reg1*2]   ; 3 cycles
        shl reg1, 4
        add reg1, reg2

by 20:  lea reg1, [reg1+reg1*4]   ; 3 cycles
        shl reg1, 2

by 21:  lea reg2, [reg1+reg1*4]   ; 3 cycles
        shl reg1, 4
        add reg1, reg2

by 22:  imul reg1, 22             ; Use the IMUL instruction.

by 23:  lea reg2, [reg1+reg1*8]   ; 3 cycles
        shl reg1, 5
        sub reg1, reg2

by 24:  lea reg1, [reg1+reg1*2]   ; 3 cycles
        shl reg1, 3

by 25:  lea reg2, [reg1+reg1*8]   ; 3 cycles
        shl reg1, 4
        add reg1, reg2

by 26:  imul reg1, 26             ; Use the IMUL instruction.

by 27:  lea reg2, [reg1+reg1*4]   ; 3 cycles
        shl reg1, 5
        sub reg1, reg2

by 28:  lea reg2, [REG1*4]        ; 3 cycles
        shl reg1, 5
        sub reg1, reg2

by 29:  lea reg2, [reg1+reg1*2]   ; 3 cycles
        shl reg1, 5
        sub reg1, reg2

by 30:  lea reg2, [reg1+reg1]     ; 3 cycles
        shl reg1, 5
        sub reg1, reg2

by 31:  mov reg2, reg1            ; 2 cycles
        shl reg1, 5
        sub reg1, reg2

by 32:  shl reg1, 5               ; 1 cycle
```

8.3 Repeated String Instructions

Optimization

Use the REP prefix judiciously when performing string operations.

Rationale

In general, using the REP prefix to repeatedly perform string instructions is less efficient than other methods, especially when copying blocks of memory. Even though using the REP prefix may seem attractive due to its small code size, a loop may yield better performance due to its minimal overhead, compared to the setup overhead of using the REP prefix. However, certain string operations can benefit from using the REP prefix when the increased throughput compared to that of a loop makes up for its setup overhead for any specific repeat count.

Guidelines for Repeated String Instructions

The following sections contain guidelines for the careful scheduling of VectorPath repeated string instructions.

Use the Largest Possible Operand Size

Always move data using the largest operand size possible. For example, in 32-bit applications, use REP MOVSD rather than REP MOVSW, and REP MOVSW rather than REP MOVSB. Use REP STOSD rather than REP STOSW, and REP STOSW rather than REP STOSB.

In 64-bit mode, a quadword data size is available and offers better performance (for example, REP MOVSQ and REP STOSQ).

Make Sure that DF is 0 (Increment)

Some string instructions with DF = 1 (decrement) may be slower.

Align Source and Destination with Operand Size

Make sure that accesses are aligned and handle the end case separately, if necessary. If there are both a source (read from) and a destination (written to) and only one can be aligned, align the destination and leave the source misaligned in order to optimize internal resources usage.

Inline REP String with Constant Small Counts

If the repeat count is constant and low (less than eight), expand REP string instructions into equivalent sequences of simple AMD64 instructions. For example, use an inline sequence of loads and stores to emulate REP MOVS or use a sequence of stores to emulate REP STOS. This technique eliminates the setup overhead of REP instructions and increases instruction throughput.

Use REP String with Constant Large Counts

If the repeat count is constant and large (in the hundreds), use REP string instructions up to approximately the data cache size. Above this limit, other techniques must be used to achieve optimal performance.

Use a Loop for REP String with Low Variable Counts

If the repeat count is variable, but is (likely) less than eight, use a simple loop to move or store the data. Otherwise, use an unrolled loop to move or store the data. These techniques avoid the overhead of REP MOVS and REP STOS.

Use a Loop for REP MOVS/CMPS If There Can Be Conflicts

The REP MOVS and REP CMPS instructions both issue two data cache operations per iteration. If certain bits of the linear addresses match, the load-store unit might have to cancel an operation and retry. To avoid this behavior, make sure the following bits in the linear address do not match:

- [6:4]—if these bits match, a cache bank conflict will occur
- [11:3]—if these bits match, a store-to-load forwarding mismatch will occur

For details, see “Store-to-Load Forwarding Restrictions” on page 74 and “L1 Data Cache Bank Conflicts” on page 90.

All Other Cases

For all other cases, it is best to call the appropriate routines in the run-time library, assuming that optimized routines are available. For more details on writing routines using repeated string instructions, see “Memory and String Routines” on page 92.

8.4 Using XOR to Clear Integer Registers

Optimization

To clear an integer register to all zeros, use the XOR instruction to exclusive OR the register with itself, as shown below.

Rationale

AMD Family 10h and 12h processors are able to avoid the false read dependency on the XOR instruction.

Examples

Acceptable

```
mov reg, 0
```

Preferred

```
xor reg, reg
```

8.5 Efficient 64-Bit Integer Arithmetic in 32-Bit Mode

Optimization

The following section contains a collection of code snippets and subroutines showing the efficient implementation of 64-bit arithmetic in 32-bit mode. Note that these are 32-bit recommendations, in 64-bit mode it is important to use 64-bit integer instructions for best performance.

Addition, subtraction, negation, and shifting are best handled by inline code. Multiplication, division, and the computation of remainders are less common operations and are usually implemented as subroutines. If these subroutines are used often, the programmer should consider inlining them. Except for division and remainder calculations, the following code works for both signed and unsigned integers. The division and remainder code shown works for unsigned integers, but can easily be extended to handle signed integers.

64-Bit Addition

```
; Add ECX:EBX to EDX:EAX, and place sum in EDX:EAX.
add eax, ebx
adc edx, ecx
```

64-Bit Subtraction

```
; Subtract ECX:EBX from EDX:EAX and place difference in EDX:EAX.
sub eax, ebx
sbb edx, ecx
```

64-Bit Negation

```
; Negate EDX:EAX.
not edx
neg eax
sbb edx, -1   ; Fix: Increment high word if low word was 0.
```

64-Bit Left Shift

```
; Shift EDX:EAX left, ??shift count in ECX (count
;  applied modulo 64).
   shld edx, eax, cl   ; First apply shift count.
   shl  eax, cl        ; ??mod 32 to EDX:EAX
   test ecx, 32        ; Need to shift by another 32?
   jz   lshift_done    ; No, done.
   mov  edx, eax       ; Left shift EDX:EAX
   xor  eax, eax       ;  by 32 bits

lshift_done:
```

64-Bit Right Shift

```
   shrd eax, edx, cl   ; First apply shift count.
   shr  edx, cl        ; ??mod 32 to EDX:EAX
```

```
  test ecx, 32       ; Need to shift by another 32?
  jz   rshift_done   ; No, done.
  mov  eax, edx      ; Left shift EDX:EAX
  xor  edx, edx      ;  by 32 bits.

rshift_done:
```

64-Bit Multiplication

```
; _llmul computes the low-order half of the product of its
;  arguments, two 64-bit integers.
;
; In:      [ESP+8]:[ESP+4] = multiplicand
;          [ESP+16]:[ESP+12] = multiplier
; Out:     EDX:EAX = (multiplicand * multiplier) % 2^64
; Destroys: EAX, ECX, EDX, EFlags

_llmul PROC
  mov edx, [esp+8]    ; multiplicand_hi
  mov ecx, [esp+16]   ; multiplier_hi
  or  edx, ecx        ; One operand >= 2^32?
  mov edx, [esp+12]   ; multiplier_lo
  mov eax, [esp+4]    ; multiplicand_lo
  jnz twomul          ; Yes, need two multiplies.
  mul edx             ; multiplicand_lo * multiplier_lo
  ret                 ; Done, return to caller.

twomul:
  imul edx, [esp+8]         ; p3_lo = multiplicand_hi * multiplier_lo
  imul ecx, eax             ; p2_lo = multiplier_hi * multiplicand_lo
  add  ecx, edx             ; p2_lo + p3_lo
  mul  dword ptr [esp+12]   ; p1 = multiplicand_lo * multiplier_lo
  add  edx, ecx             ; p1 + p2_lo + p3_lo = result in EDX:EAX
  ret                       ; Done, return to caller.

_llmul ENDP
```

64-Bit Unsigned Division

```
; _ulldiv divides two unsigned 64-bit integers and returns the quotient.
;
; In:      [ESP+8]:[ESP+4] = dividend
;          [ESP+16]:[ESP+12] = divisor
; Out:     EDX:EAX = quotient of division
; Destroys: EAX, ECX, EDX, EFlags

_ulldiv PROC
  push ebx            ; Save EBX as per calling convention.
  mov  ecx, [esp+20]  ; divisor_hi
  mov  ebx, [esp+16]  ; divisor_lo
  mov  edx, [esp+12]  ; dividend_hi
  mov  eax, [esp+8]   ; dividend_lo
  test ecx, ecx       ; divisor > (2^32 – 1)?
  jnz  big_divisor    ; Yes, divisor > 2^32 – 1.
  cmp  edx, ebx       ; Only one division needed (ECX = 0)?
  jae  two_divs       ; Need two divisions.
  div  ebx            ; EAX = quotient_lo
```

```
  mov  edx, ecx       ; EDX = quotient_hi = 0 (quotient in EDX:EAX)
  pop  ebx            ; Restore EBX as per calling convention.
  ret                 ; Done, return to caller.

two_divs:
  mov  ecx, eax   ; Save dividend_lo in ECX.
  mov  eax, edx   ; Get dividend_hi.
  xor  edx, edx   ; Zero-extend it into EDX:EAX.
  div  ebx        ; quotient_hi in EAX
  xchg eax, ecx   ; ECX = quotient_hi, EAX = dividend_lo
  div  ebx        ; EAX = quotient_lo
  mov  edx, ecx   ; EDX = quotient_hi (quotient in EDX:EAX)
  pop  ebx        ; Restore EBX as per calling convention.
  ret             ; Done, return to caller.

big_divisor:
  push edi                ; Save EDI as per calling convention.
  mov  edi, ecx           ; Save divisor_hi.
  shr  edx, 1             ; Shift both divisor and dividend right
  rcr  eax, 1             ;  by 1 bit.
  ror  edi, 1
  rcr  ebx, 1
  bsr  ecx, ecx           ; ECX = number of remaining shifts
  shrd ebx, edi, cl       ; Scale down divisor and dividend
  shrd eax, edx, cl       ;  such that divisor is less than
  shr  edx, cl            ;  2^32 (that is, it fits in EBX).
  rol  edi, 1             ; Restore original divisor_hi.
  div  ebx                ; Compute quotient.
  mov  ebx, [esp+12]      ; dividend_lo
  mov  ecx, eax           ; Save quotient.
  imul edi, eax           ; quotient * divisor high word (??low only)
  mul  dword ptr [esp+20] ; quotient * divisor low word
  add  edx, edi           ; EDX:EAX = quotient * divisor
  sub  ebx, eax           ; dividend_lo – (quot.*divisor)_lo
  mov  eax, ecx           ; Get quotient.
  mov  ecx, [esp+16]      ; dividend_hi
  sbb  ecx, edx           ; Subtract (divisor * quot.) from dividend.
  sbb  eax, 0             ; Adjust quotient if remainder negative.
  xor  edx, edx           ; Clear high word of quot. (EAX<=FFFFFFFFh).
  pop  edi                ; Restore EDI as per calling convention.
  pop  ebx                ; Restore EBX as per calling convention.
  ret                     ; Done, return to caller.

_ulldiv ENDP
```

64-Bit Signed Division

```
; _lldiv divides two signed 64-bit numbers and delivers the quotient
;
; In:       [ESP+8]:[ESP+4] = dividend
;           [ESP+16]:[ESP+12] = divisor
; Out:      EDX:EAX = quotient of division
; Destroys: EAX, ECX,E DX, EFlags

_lldiv PROC
  push ebx    ; Save EBX as per calling convention.
  push esi    ; Save ESI as per calling convention.
  push edi    ; Save EDI as per calling convention.
  mov  ecx, [esp+28]   ; divisor_hi
  mov  ebx, [esp+24]   ; divisor_lo
  mov  edx, [esp+20]   ; dividend_hi
  mov  eax, [esp+16]   ; dividend_lo
  mov  esi, ecx        ; divisor_hi
  xor  esi, edx        ; divisor_hi ^ dividend_hi
  sar  esi, 31         ; (quotient < 0) ? -1 : 0
  mov  edi, edx        ; dividend_hi
  sar  edi, 31         ; (dividend < 0) ? -1 : 0
  xor  eax, edi        ; If (dividend < 0),
  xor  edx, edi        ;  compute 1's complement of dividend.
  sub  eax, edi        ; If (dividend < 0),
  sbb  edx, edi        ;  compute 2's complement of dividend.
  mov  edi, ecx        ; divisor_hi
  sar  edi, 31         ; (divisor < 0) ? -1 : 0
  xor  ebx, edi        ; If (divisor < 0),
  xor  ecx, edi        ;  compute 1's complement of divisor.
  sub  ebx, edi        ; If (divisor < 0),
  sbb  ecx, edi        ;  compute 2's complement of divisor.
  jnz  big_divisor     ; divisor > 2^32 - 1
  cmp  edx, ebx        ; Only one division needed (ECX = 0)?
  jae  two_divs        ; Need two divisions.
  div  ebx             ; EAX = quotient_lo
  mov  edx, ecx        ; EDX = quotient_hi = 0 (quotient in EDX:EAX)
  xor  eax, esi        ; If (quotient < 0),
  xor  edx, esi        ;  compute 1's complement of result.
  sub  eax, esi        ; If (quotient < 0),
  sbb  edx, esi        ;  compute 2's complement of result.
  pop  edi             ; Restore EDI as per calling convention.
  pop  esi             ; Restore ESI as per calling convention.
  pop  ebx             ; Restore EBX as per calling convention.
  ret                ; Done, return to caller.

two_divs:
  mov  ecx, eax     ; Save dividend_lo in ECX.
  mov  eax, edx     ; Get dividend_hi.
  xor  edx, edx     ; Zero-extend it into EDX:EAX.
  div  ebx          ; quotient_hi in EAX
  xchg eax, ecx     ; ECX = quotient_hi, EAX = dividend_lo
  div  ebx          ; EAX = quotient_lo
  mov  edx, ecx     ; EDX = quotient_hi (quotient in EDX:EAX)
  jmp  make_sign    ; Make quotient signed.
```

```
big_divisor:
   sub  esp, 12            ; Create three local variables.
   mov  [esp], eax          ; dividend_lo
   mov  [esp+4], ebx        ; divisor_lo
   mov  [esp+8], edx        ; dividend_hi
   mov  edi, ecx            ; Save divisor_hi.
   shr  edx, 1             ; Shift both
   rcr  eax, 1             ;  divisor and
   ror  edi, 1             ;  and dividend
   rcr  ebx, 1             ;  right by 1 bit.
   bsr  ecx, ecx            ; ECX = number of remaining shifts
   shrd ebx, edi, cl        ; Scale down divisor and
   shrd eax, edx, cl        ;  dividend such that divisor is
   shr  edx, cl            ;  less than 2^32 (that is, fits in EBX).
   rol  edi, 1             ; Restore original divisor_hi.
   div  ebx                ; Compute quotient.
   mov  ebx, [esp]          ; dividend_lo
   mov  ecx, eax            ; Save quotient.
   imul edi, eax           ; quotient * divisor high word (??low only)
   mul  DWORD PTR [esp+4]   ; quotient * divisor low word
   add  edx, edi           ; EDX:EAX = quotient * divisor
   sub  ebx, eax           ; dividend_lo - (quot.*divisor)_lo
   mov  eax, ecx            ; Get quotient.
   mov  ecx, [esp+8]        ; dividend_hi
   sbb  ecx, edx           ; Subtract (divisor * quot.) from dividend
   sbb  eax, 0             ; Adjust quotient if remainder is negative.
   xor  edx, edx           ; Clear high word of quotient.
   add  esp, 12            ; Remove local variables.

make_sign:
   xor eax, esi   ; If (quotient < 0),
   xor edx, esi   ;  compute 1's complement of result.
   sub eax, esi   ; If (quotient < 0),
   sbb edx, esi   ;  compute 2's complement of result.
   pop edi        ; Restore EDI as per calling convention.
   pop esi        ; Restore ESI as per calling convention.
   pop ebx        ; Restore EBX as per calling convention.
   ret           ; Done, return to caller.
_lldiv ENDP
```

64-Bit Unsigned Remainder Computation

```
; _ullrem divides two unsigned 64-bit integers and returns the remainder.
;
; In:       [ESP+8]:[ESP+4] = dividend
;           [ESP+16]:[ESP+12] = divisor
;
; Out:      EDX:EAX = remainder of division
;
; Destroys: EAX, ECX, EDX, EFlags

_ullrem PROC
   push ebx             ; Save EBX as per calling convention.
   mov  ecx, [esp+20]    ; divisor_hi
   mov  ebx, [esp+16]    ; divisor_lo
   mov  edx, [esp+12]    ; dividend_hi
   mov  eax, [esp+8]     ; dividend_lo
```

```
   test ecx, ecx        ; divisor > 2^32 - 1?
   jnz  r_big_divisor   ; Yes, divisor > 32^32 - 1.
   cmp  edx, ebx        ; Only one division needed (ECX = 0)?
   jae  r_two_divs      ; Need two divisions.
   div  ebx             ; EAX = quotient_lo
   mov  eax, edx        ; EAX = remainder_lo
   mov  edx, ecx        ; EDX = remainder_hi = 0
   pop  ebx             ; Restore EBX per calling convention.
   ret                  ; Done, return to caller.

r_two_divs:
   mov ecx, eax   ; Save dividend_lo in ECX.
   mov eax, edx   ; Get dividend_hi.
   xor edx, edx   ; Zero-extend it into EDX:EAX.
   div ebx        ; EAX = quotient_hi, EDX = intermediate remainder
   mov eax, ecx   ; EAX = dividend_lo
   div ebx        ; EAX = quotient_lo
   mov eax, edx   ; EAX = remainder_lo
   xor edx, edx   ; EDX = remainder_hi = 0
   pop ebx        ; Restore EBX as per calling convention.
   ret            ; Done, return to caller.

r_big_divisor:
   push edi             ; Save EDI as per calling convention.
   mov  edi, ecx        ; Save divisor_hi.
   shr  edx, 1          ; Shift both divisor and dividend right
   rcr  eax, 1          ;  by 1 bit.
   ror  edi, 1
   rcr  ebx, 1
   bsr  ecx, ecx        ; ECX = number of remaining shifts
   shrd ebx, edi, cl    ; Scale down divisor and dividend such
   shrd eax, edx, cl    ;  that divisor is less than 2^32
   shr  edx, cl         ;  (that is, it fits in EBX).
   rol  edi, 1          ; Restore original divisor (EDI:ESI).
   div  ebx             ; Compute quotient.
   mov  ebx, [esp+12]   ; dividend low word
   mov  ecx, eax        ; Save quotient.
   imul edi, eax        ; quotient * divisor high word (??low only)
   mul  DWORD PTR [esp+20]  ; quotient * divisor low word
   add  edx, edi        ; EDX:EAX = quotient * divisor
   sub  ebx, eax        ; dividend_lo – (quot.*divisor)_lo
   mov  ecx, [esp+16]   ; dividend_hi
   mov  eax, [esp+20]   ; divisor_lo
   sbb  ecx, edx        ; Subtract divisor * quot. from dividend.
   sbb  edx, edx        ; (remainder < 0) ? 0xFFFFFFFF : 0
   and  eax, edx        ; (remainder < 0) ? divisor_lo : 0
   and  edx, [esp+24]   ; (remainder < 0) ? divisor_hi : 0
   add  eax, ebx        ; remainder += (remainder < 0) ? divisor : 0
   pop  edi             ; Restore EDI as per calling convention.
   pop  ebx             ; Restore EBX as per calling convention.
   ret                  ; Done, return to caller.

_ullrem ENDP
```

64-Bit Signed Remainder Computation

```
; _llrem divides two signed 64-bit numbers and returns the remainder.
;
; In:       [ESP+8]:[ESP+4] = dividend
;           [ESP+16]:[ESP+12] = divisor
;
; Out:      EDX:EAX = remainder of division
;
; Destroys: EAX, ECX, EDX, EFlags

   push ebx              ; Save EBX as per calling convention.
   push esi              ; Save ESI as per calling convention.
   push edi              ; Save EDI as per calling convention.
   mov  ecx, [esp+28]    ; divisor-hi
   mov  ebx, [esp+24]    ; divisor-lo
   mov  edx, [esp+20]    ; dividend-hi
   mov  eax, [esp+16]    ; dividend-lo
   mov  esi, edx         ; sign(remainder) == sign(dividend)
   sar  esi, 31          ; (remainder < 0) ? -1 : 0
   mov  edi, edx         ; dividend-hi
   sar  edi, 31          ; (dividend < 0) ? -1 : 0
   xor  eax, edi         ; If (dividend < 0),
   xor  edx, edi         ;  compute 1's complement of dividend.
   sub  eax, edi         ; If (dividend < 0),
   sbb  edx, edi         ;  compute 2's complement of dividend.
   mov  edi, ecx         ; divisor-hi
   sar  edi, 31          ; (divisor < 0) ? -1 : 0
   xor  ebx, edi         ; If (divisor < 0),
   xor  ecx, edi         ;  compute 1's complement of divisor.
   sub  ebx, edi         ; If (divisor < 0),
   sbb  ecx, edi         ;  compute 2's complement of divisor.
   jnz  sr_big_divisor   ; divisor > 2^32 - 1
   cmp  edx, ebx         ; Only one division needed (ECX = 0)?
   jae  sr_two_divs      ; No, need two divisions.
   div  ebx              ; EAX = quotient_lo
   mov  eax, edx         ; EAX = remainder_lo
   mov  edx, ecx         ; EDX = remainder_lo = 0
   xor  eax, esi         ; If (remainder < 0),
   xor  edx, esi         ;  compute 1's complement of result.
   sub  eax, esi         ; If (remainder < 0),
   sbb  edx, esi         ;  compute 2's complement of result.
   pop  edi              ; Restore EDI as per calling convention.
   pop  esi              ; Restore ESI as per calling convention.
   pop  ebx              ; Restore EBX as per calling convention.
   ret                   ; Done, return to caller.

sr_two_divs:
   mov ecx, eax          ; Save dividend_lo in ECX.
   mov eax, edx          ; Get dividend_hi.
   xor edx, edx          ; Zero-extend it into EDX:EAX.
   div ebx               ; EAX = quotient_hi, EDX = intermediate remainder
   mov eax, ecx          ; EAX = dividend_lo
   div  ebx              ; EAX = quotient_lo
   mov  eax, edx         ; remainder_lo
   xor  edx, edx         ; remainder_hi = 0
   jmp  sr_makesign      ; Make remainder signed.
```

```
sr_big_divisor:
   sub  esp, 16            ; Create three local variables.
   mov  [esp], eax          ; dividend_lo
   mov  [esp+4], ebx        ; divisor_lo
   mov  [esp+8], edx        ; dividend_hi
   mov  [esp+12], ecx       ; divisor_hi
   mov  edi, ecx           ; Save divisor_hi.
   shr  edx, 1             ; Shift both
   rcr  eax, 1             ;  divisor and
   ror  edi, 1             ;  and dividend
   rcr  ebx, 1             ;  right by 1 bit.
   bsr  ecx, ecx           ; ECX = number of remaining shifts
   shrd ebx, edi, cl        ; Scale down divisor and
   shrd eax, edx, cl        ;  dividend such that divisor is
   shr  edx, cl            ;  less than 2^32 (that is, fits in EBX).
   rol  edi, 1             ; Restore original divisor_hi.
   div  ebx                ; Compute quotient.
   mov  ebx, [esp]          ; dividend_lo
   mov  ecx, eax           ; Save quotient.
   imul edi, eax           ; quotient * divisor high word (??low only)
   mul  DWORD PTR [esp+4]   ; quotient * divisor low word
   add  edx, edi           ; EDX:EAX = quotient * divisor
   sub  ebx, eax           ; dividend_lo - (quot.*divisor)_lo
   mov  ecx, [esp+8]        ; dividend_hi
   sbb  ecx, edx           ; Subtract divisor * quot. from dividend.
   sbb  eax, eax           ; remainder < 0 ? 0xffffffff : 0
   mov  edx, [esp+12]       ; divisor_hi
   and  edx, eax           ; remainder < 0 ? divisor_hi : 0
   and  eax, [esp+4]        ; remainder < 0 ? divisor_lo : 0
   add  eax, ebx           ; remainder_lo
   add  edx, ecx           ; remainder_hi
   add  esp, 16            ; Remove local variables.

sr_makesign:
   xor eax, esi   ; If (remainder < 0),
   xor edx, esi   ;  compute 1's complement of result.
   sub eax, esi   ; If (remainder < 0),
   sbb edx, esi   ;  compute 2's complement of result.
   pop edi        ; Restore EDI as per calling convention.
   pop esi        ; Restore ESI as per calling convention.
   pop ebx        ; Restore EBX as per calling convention.
   ret            ; Done, return to caller.
```

8.6 Derivation of Algorithm, Multiplier, and Shift Factor for Integer Division by Constants

The following examples illustrate the derivation of algorithm, multiplier and shift factor for signed and unsigned integer division.

Unsigned Integer Division

The utility udiv.exe was compiled from the code shown in this section. The utilities provided in this document are for reference only and are not supported by AMD.

The following code derives the multiplier value used when performing integer division by constants. The code works for unsigned integer division and for odd divisors between 1 and $2^{31} - 1$, inclusive. For divisors of the form $d = d' * 2n$, the multiplier is the same as for d' and the shift factor is $s + n$.

Example

```
/* This program determines the algorithm (a), multiplier (m), and
   shift factor (s) to be used to accomplish *unsigned* division by
   a constant divisor. Compile with MSVC.
*/

#include <stdio.h>

typedef unsigned __int64 U64;
typedef unsigned long    U32;

U32 log2(U32 i)
{
  U32 t = 0;
  i = i >> 1;
  while (i) {
    i = i >> 1;
    t++;
  }
  return(t);
}

U32 res1, res2;
U32 d, l, s, m, a, r, n, t;
U64 m_low, m_high, j, k;

int main (void)
{
  fprintf(stderr, "\n");
  fprintf(stderr, "Unsigned division by constant\n");
  fprintf(stderr, "=============================\n\n");
  fprintf(stderr, "enter divisor: ");
  scanf("%lu", &d);
  printf("\n");
  if (d == 0) goto printed_code;

  if (d >= 0x80000000UL) {
    printf("; dividend: register or memory location\n");
    printf("\n");
    printf("CMP    dividend, 0%08lXh\n", d);
    printf("MOV    EDX, 0\n");
    printf("SBB    EDX, -1\n");
    printf("\n");
    printf("; quotient now in EDX\n");
```

```
  goto printed_code;
}

/* Reduce divisor until it becomes odd. */

n = 0;
t = d;
while (!(t & 1)) {
  t >>= 1;
  n++;
}

if (t == 1) {
  if (n == 0) {
    printf("; dividend: register or memory location\n");
    printf("\n");
    printf("MOV    EDX, dividend\n", n);
    printf("\n");
    printf("; quotient now in EDX\n");
  }
  else {
    printf("; dividend: register or memory location\n");
    printf("\n");
    printf("SHR    dividend, %d\n", n);
    printf("\n");
    printf("; quotient replaced dividend\n");
  }
  goto printed_code;
}

/* Generate m, s for algorithm 0. Based on: Granlund, T.; Montgomery,
   P.L.: "Division by Invariant Integers using Multiplication."
   SIGPLAN Notices, Vol. 29, June 1994, page 61.
*/

l = log2(t) + 1;
j = (((U64)(0xffffffff)) % ((U64)(t)));
k = (((U64)(1)) << (32 + l)) / ((U64)(0xffffffff - j));
m_low = (((U64)(1)) << (32 + l)) / t;
m_high = ((((U64)(1)) << (32 + l)) + k) / t;
while (((m_low >> 1) < (m_high >> 1)) && (l > 0)) {
  m_low = m_low >> 1;
  m_high = m_high >> 1;
  l = l - 1;
}
if ((m_high >> 32) == 0) {
  m = ((U32)(m_high));
  s = l;
  a = 0;
}
  /* Generate m and s for algorithm 1. Based on: Magenheimer, D.J.; et al:
  "Integer Multiplication and Division on the HP Precision Architecture."
  IEEE Transactions on Computers, Vol. 37, No. 8, August 1988, page 980.*/
else {
  s = log2(t);
  m_low = (((U64)(1)) << (32 + s)) / ((U64)(t));
```

```
    r = ((U32)((((U64)(1)) << (32 + s)) % ((U64)(t))));
    m = (r < ((t >> 1) + 1)) ? ((U32)(m_low)) : ((U32)(m_low)) + 1;
    a = 1;
  }
    /* Reduce multiplier for either algorithm to smallest possible.*/
 while (!(m & 1)) {
    m = m >> 1;
    s--;
  }

  /* Adjust multiplier for reduction of even divisors. */

  s += n;

  if (a) {
    printf("; dividend: register other than EAX or memory location\n");
    printf("\n");
    printf("MOV    EAX, 0%08lXh\n", m);
    printf("MUL    dividend\n");
    printf("ADD    EAX, 0%08lXh\n", m);
    printf("ADC    EDX, 0\n");
    if (s) printf("SHR    EDX, %d\n", s);
    printf("\n");
    printf("; quotient now in EDX\n");
  }
  else {
    printf("; dividend: register other than EAX or memory location\n");
    printf("\n");
    printf("MOV    EAX, 0%08lXh\n", m);
    printf("MUL    dividend\n");
    if (s) printf("SHR    EDX, %d\n", s);
    printf("\n");
    printf("; quotient now in EDX\n");
  }

printed_code:

  fprintf(stderr, "\n");
  exit(0);

  return(0);
}
```

Signed Integer Division

The utility sdiv.exe was compiled using the following code. The utilities provided in this document are for reference only and are not supported by AMD.

Example

```
/* This program determines the algorithm (a), multiplier (m), and
   shift factor (s) to be used to accomplish *signed* division by
   a constant divisor. Compile with MSVC.
*/
```

```
#include <stdio.h>

typedef unsigned __int64 U64;
typedef unsigned long    U32;

U32 log2(U32 i)
{
  U32 t = 0;
  i = i >> 1;
  while (i) {
    i = i >> 1;
    t++;
  }
  return(t);
}

long e;
U32 res1, res2;
U32 oa, os, om;
U32 d, l, s, m, a, r, t;
U64 m_low, m_high, j, k;

int main(void)

{
  fprintf(stderr, "\n");
  fprintf(stderr, "Signed division by constant\n");
  fprintf(stderr, "============================\n\n");

  fprintf(stderr, "enter divisor: ");
  scanf("%ld", &d);
  fprintf(stderr, "\n");

  e = d;
  d = labs(d);

  if (d == 0) goto printed_code;

  if (e == (-1)) {
    printf("; dividend: register or memory location\n");
    printf("\n");
    printf("NEG    dividend\n");
    printf("\n");
    printf("; quotient replaced dividend\n");
    goto printed_code;
  }
  if (d == 2) {
    printf("; dividend expected in EAX\n");
    printf("\n");
    printf("CMP    EAX, 080000000h\n");
    printf("SBB    EAX, -1\n");
    printf("SAR    EAX, 1\n");
    if (e < 0) printf("NEG    EAX\n");
    printf("\n");
    printf("; quotient now in EAX\n");
```

```
    goto printed_code;
  }

  if (!(d & (d - 1))) {
    printf("; dividend expected in EAX\n");
    printf("\n");
    printf("CDQ\n");
    printf("AND    EDX, 0%08lXh\n", (d-1));
    printf("ADD    EAX, EDX\n");
    if (log2(d)) printf("SAR    EAX, %d\n", log2(d));
    if (e < 0)   printf("NEG    EAX\n");
    printf("\n");
    printf("; quotient now in EAX\n");
    goto printed_code;
  }

  /* Determine algorithm (a), multiplier (m), and shift factor (s) for 32-bit
     signed integer division. Based on: Granlund, T.; Montgomery, P.L.:
     "Division by Invariant Integers using Multiplication". SIGPLAN Notices,
     Vol. 29, June 1994, page 61.
  */

  l = log2(d);
  j = (((U64)(0x80000000)) % ((U64)(d)));
  k = (((U64)(1)) << (32 + l)) / ((U64)(0x80000000 - j));
  m_low = (((U64)(1)) << (32 + l)) / d;
  m_high = ((((U64)(1)) << (32 + l)) +  k) / d;



  while (((m_low >> 1) < (m_high >> 1)) && (l > 0)) {
    m_low  = m_low  >> 1;
    m_high = m_high >> 1;
    l = l - 1;
  }
  m = ((U32)(m_high));
  s = l;
  a = (m_high >> 31) ? 1 : 0;

  if (a) {
    printf("; dividend: memory location or register other than EAX or EDX\n");
    printf("\n");
    printf("MOV    EAX, 0%08LXh\n", m);
    printf("IMUL   dividend\n");
    printf("MOV    EAX, dividend\n");
    printf("ADD    EDX, EAX\n");
    if (s) printf("SAR    EDX, %d\n", s);
    printf("SHR    EAX, 31\n");
    printf("ADD    EDX, EAX\n");
    if (e < 0) printf("NEG    EDX\n");
    printf("\n");
    printf("; quotient now in EDX\n");
  }
  else {
    printf("; dividend: memory location of register other than EAX or EDX\n");
```

```
    printf("\n");
    printf("MOV    EAX, 0%08LXh\n", m);
    printf("IMUL   dividend\n");
    printf("MOV    EAX, dividend\n");
    if (s) printf("SAR    EDX, %d\n", s);
    printf("SHR    EAX, 31\n");
    printf("ADD    EDX, EAX\n");
    if (e < 0) printf("NEG    EDX\n");
    printf("\n");
    printf("; quotient now in EDX\n");
  }

printed_code:

  fprintf(stderr, "\n");
  exit(0);
}
```

8.7 Optimizing Integer Division

Optimization

This section on Optimizing Integer Division applies to AMD Family 10h processors only.

For all data types, except in 8-bit division, making the absolute value of the most significant word (in DX/EDX/RDX) of the dividend all 0s for the DIV instruction or all 0s or all 1s for the IDIV instruction lowers the latency of integer division. If this is not possible, then use a smaller data type for integer division.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Integer division latency is dependent on the operand size. These latency numbers could go down even lower, depending on the number of leading zero bits in the absolute value of the dividend. Table 7 provides details about the latency of any particular instance of a DIV/IDIV instruction.

When integer division constitutes a substantial computational load, it may be beneficial to check whether the most significant word of the absolute value of the dividend in DX/EDX/RDX can be set to all 0s for DIV or to all 0s or all 1s for IDIV. If that is not possible, then using a smaller division size will help to lower the latency.

In any case, assembly language output generated by high-level language compilers should be verified that the desired code is generated. When dividing by a constant, if possible, substitute the division with a multiplication. (See “Replacing Division with Multiplication” on page 121 for more details.)

Table 7. DIV/IDIV Latencies

Divisor	Absolute Value of Dividend	Latency	
		DIV	IDIV
8 Bits	Reg	17	18
	Mem	16	20
16, 32, 64 Bits	0	18	22
16 Bits	> 0 and < 2^{16}	18 (MSB in bit 0, 1, or 2); or 15 + bit position of the MSB of the dividend (MSB >= bit 3)	22 + bit position of the MSB of the absolute value of the dividend
32 Bits	> 0 and < 2^{32}		
64 Bits	> 0 and < 2^{64}		
16 Bits	≥ 2^{16}	24	31
32 Bits	≥ 2^{32}	40	47
64 Bits	≥ 2^{64}	72	79
Note: *MSB—Most significant bit.*			

Table 8. Integer Hardware Divider Latency (12H only)

Divisor	Latency Range of DIV and IDIV
8 Bits	9 – 13 cycles
16 Bits	9 – 17 cycles
32 Bits	9 – 25 cycles
64 Bits	9 – 41 cycles
Note: *The quotient will be available two cycles earlier than the latency specified above.*	

8.8 Efficient Implementation of Population Count and Leading-Zero Count

Optimization

Use the POPCNT instruction to implement a population count and use LZCNT to perform a leading-zero count operation.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

A population count determines the number of set bits in a bit string. The POPCNT instruction, a new instruction for AMD Family 10h and 12h processors, is the preferred way to implement a population count.

A leading-zero count is an operation that counts the number of leading bits in the input operand that are cleared to zero. Counting starts downward from the most significant bit and stops at the highest bit which is one or when the least significant bit is encountered. LZCNT is a new instruction for AMD Family 10h and 12h processors that implement this function.

The POPCNT and LZCNT instructions can count the bits in a 32-bit operand in 32-bit mode or a 64-bit operand in 64-bit mode.

Chapter 9 Optimizing with SIMD Instructions

The 64-bit and 128-bit SIMD instructions—SSE, SSE2, SSE3, SSE4a instructions—should be used to encode floating-point and packed integer operations.

- The SIMD instructions use a flat register file rather than the stack register file used by x87 floating-point instructions. This allows arbitrary sequences of operations to map more efficiently to the instruction set.
- AMD Family 10h and 12h processors with 128-bit multipliers and adders achieve better throughput using SSE, SSE2, SSE3, and SSE4a instructions. (Double precision throughput is 2× and single precision is 4× the throughput of x87.)
- SSE, SSE2, SSE3, and SSE4a instructions work well in both 32-bit and 64-bit threads.
- In 64-bit mode, there are twice as many XMM registers available as in 32-bit mode, however, the number of x87 registers is the same in both 32-bit mode and 64-bit mode.

The SIMD instructions provide a theoretical single-precision peak throughput of four additions and four multiplications per clock cycle, whereas x87 instructions can only sustain one addition and one multiplication per clock cycle. The double-precision peak throughput of the SSE, SSE2, SSE3, and SSE4a instructions is two additions and two multiplications per clock cycle.

This chapter covers the following topics:

Topic	Page
Ensure All Packed Floating-Point Data are Aligned	148
Explicit Load Instructions	148
Unaligned and Aligned Data Access	149
Moving Data Between General-Purpose and MMX™ or XMM Registers	149
Use SSE Instructions to Construct Fast Block-Copy Routines in 32-Bit Mode or 64-bit Mode	150
EMMS Usage	151
Using SIMD Instructions for Fast Square Roots and Divisions	152
Use XOR Operations to Negate Operands of SSEx Instructions	154
Clearing MMX™ and XMM Registers with XOR Instructions	155
Finding the Floating-Point Absolute Value of Operands of SSE and SSE2 Instructions	156
Accumulating Single-Precision Floating-Point Numbers Using SSE and SSE2 Instructions	157
Complex-Number Arithmetic Using SSE, SSE2, and SSE3 Instructions	158
Optimized 4 X 4 Matrix Multiplication on 4 X 1 Column Vector Routines	164
Floating-Point-to-Integer Conversion	167
Reuse of Dead Registers	167
Floating-Point Scalar Conversions	168
Scalar Merge Optimization	172

9.1 Ensure All Packed Floating-Point Data are Aligned

Optimization

Align all packed floating-point data on 16-byte boundaries.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Misaligned memory accesses reduce the available memory bandwidth and SSE, SSE2, and SSE3 instructions have shorter latencies when operating on aligned memory operands.

Aligning data on 16-byte boundaries reduces the possibility of stalling floating-point addition and multiplication instructions that are dependent on the load data. See also section 9.3, “Unaligned and Aligned Data Access” on page 149.

9.2 Explicit Load Instructions

Optimization

Use `MOVSD xmm1, mem64` when loading a scalar floating-point double-precision value from memory. Use `MOVSS xmm1, mem32` when loading a scalar floating-point single-precision value from memory.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The `MOVSD xmm1, mem64` instruction is more efficient than `MOVLPD xmm1, mem64` on an AMD Family 10h and 12h processor, since it modifies the entire XMM register, thus breaking the dependency chain on the high-order bits of the register.

The `MOVSS xmm1, mem32` instruction zeroes the unaffected remaining bits of the XMM register and breaks any dependency chain. It also assures that the upper half of the XMM register contains a normal floating-point single-precision value.

9.3 Unaligned and Aligned Data Access

Optimization

When data alignment cannot be guaranteed, use MOVUP*x* or MOVDQU for loads and the MOVLP*x*/MOVHP*x* pair for stores on AMD Family 10h and 12h processors.

Otherwise, when data alignment is guaranteed, always use MOVAP*x* or MOVDQA.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

On AMD Family 10h and 12h processors, the MOVUP*x* and MOVDQU instructions are DirectPath for loads, but VectorPath for stores, therefore the MOVLP*x* / MOVHP*x* pair should be used for stores. MOVUP*x* or MOVDQU loads can be as fast as MOVAP*x* or MOVDQA loads when the memory location is 16-byte aligned. The MOVUP*x* and MOVDQU instructions break dependency chains by changing the entire XMM register when loading data from memory. On the other hand, because both MOVLP*x* and MOVHP*x* loads change one half of the XMM register, there is a dependency between each of them and any previous instructions that change any part of the same XMM register.

9.4 Moving Data Between General-Purpose and MMX™ or XMM Registers

Optimization

When moving data from a GPR to an MMX or XMM register, use separate store and load instructions to move the data first from the source register to a temporary location in memory and then from memory into the destination register, taking the memory latency into account when scheduling both stages of the load-store sequence.

When moving data from an MMX or XMM register to a general-purpose register, use the MOVD instruction.

Whenever possible, use loads and stores of the same data length. (See 5.3, ‘Store-to-Load Forwarding Restrictions” on page 74 for more information.)

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

When a GPR is the *source* to the MOVD instruction, MOVD is a higher-latency DirectPath Double instruction; compared to the low-latency DirectPath Single instructions used first to store the contents of the GPR to memory and then to load this value into an MMX or XMM register.

When a GPR is the *destination* of MOVD, MOVD is a DirectPath Single instruction.

9.5 Use SSE Instructions to Construct Fast Block-Copy Routines in 32-Bit Mode or 64-bit Mode

Optimization

Use XMM registers instead of general purpose registers to copy blocks of data that reside in cache.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

An SSE load can read 16 bytes in a single clock cycle, while an SSE store can write 16 bytes in 2 cycles. MOVDQU can safely access 16-byte SSE data regardless of alignment, with performance equal to MOVDQA when data is actually 16-byte aligned, so use MOVDQU and align the destination and/or the source to 16-byte boundaries when possible.

Example

The following code illustrates an implementation of an optimized memory block copy.

Note: *The loop is unrolled to use two XMM registers, to hide the execution latencies of the pointer/counter arithmetic and the branch.*

```
mov rcx, [destination]  ; 16-byte aligned, if possible
mov rdx, [source]       ; 16-byte aligned, if possible
mov rax, [count]        ; make sure it's at least 32 bytes!
```

```
   sshr rax, 5       ; we move 32 bytes per loop
   jz SSE_done

   align 16         ; align loop top for best performance
SSE_loop:
   movdqu xmm0, [rdx]
   movdqu xmm1, [rdx + 16]
   add rdx, 32
   movlpd [rcx], xmm0
   movhpd [rcx+8], xmm0
   movlpd [rcx+16], xmm1
   movhpd [rcx+24], xmm1
   add rcx, 32
   dec rax
   jnz SSE_loop

SSE_done:
   (move any residual bytes)
```

9.6 EMMS Usage

Optimization

Use EMMS to clean up the register file between an x87 instruction and a following MMX instruction or vice versa.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Use the EMMS instruction when switching between the x87 floating-point unit and MMX instructions. The EMMS instruction is a fast low-latency instruction in AMD Family 10h processors.

x87 and MMX instructions share the same architectural registers, so there is no easy way to use them concurrently without cleaning up the register file in between by using the EMMS instruction. For more information, see the *AMD64 Architecture Programmer's Manual, Volume 1: Application Programming*, order# 24592, and *AMD64 Architecture Programmer's Manual, Volume 5: 64-Bit Media and x87 Floating-Point Instructions*, order# 26569.

9.7 Using SIMD Instructions for Fast Square Roots and Divisions

Optimization

Use SIMD vectorized square root (SQRTSS/SQRTPS) and reciprocal (RCPSS/RCPPS) instructions to calculate square roots and divisions of single-precision numbers.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The calculation of reciprocal square root and reciprocation of single-precision numbers are often used in multimedia applications. These SIMD instructions can be used for such operations when a slight inaccuracy is acceptable.

Although these instructions return their results with a maximum error of 2^{-11}, they can be used with the Newton-Raphson method to obtain more accurate results.

For square roots accurate to 2.5 ULPs, the following algorithm is obtained after one Newton-Raphson iteration:

```
y = 0.5 * a * x * (3.0 - a * x * x)
```

Where *x* is the initial approximation of the reciprocal of the square root of *a* and *y*, the square root of *a*.

For divisions accurate to 1.5 ULPs, the following algorithm is obtained after one Newton-Raphson iteration:

```
y = a * x * (2.0 - b * x)
```

Where *x* is the initial approximation of the reciprocal of *b* and *y*, the quotient of *a* divided by *b*.

Although more Newton-Raphson iterations could be used to increase accuracy, the execution time would be longer than the equivalent instructions. This implementation of the Newton-Raphson technique is not 100% compliant to the IEEE-754 specification, but its results are acceptable in most applications.

Example

The following functions calculate the square root:

```
#include <xmmintrin.h>
```

```
/* nr_sqrtf: return scalar square root accurate to 2.5ulps.

    This approximation assumes finite math; never returns denormals, but zero;
    does not return the expected values after C89;
    is not compliant with IEEE754 semantics. */

float nr_sqrtf (float a)
{
  __m128 x0, x1, x2, x3, x4, x5, m;
  float y;

  m = _mm_cmpneq_ss (_mm_set_ss (a), _mm_setzero_ps ()); // m = (a != 0.0? T: F)

  x0 = _mm_rsqrt_ss (_mm_set_ss (a));                    // x0 = initial estimate
  x1 = _mm_and_ps (m, x0);                               // x1 = m & x0
  x2 = _mm_mul_ss (_mm_set_ss (a), x1);                  // x2 = a * x1
  x3 = _mm_mul_ss (_mm_set_ss (0.5F), x2);               // x3 = 0.5 * x2
  x4 = _mm_mul_ss (x1, x2);                              // x4 = x1 * x2
  x5 = _mm_sub_ss (_mm_set_ss (3.0F), x4);               // x5 = 3.0 - x4

  _mm_store_ss (&y, _mm_mul_ss (x3, x5));                // y = x3 * x5
  return (y);                                            // y = sqrtf (a)
}

/* nr_sqrtvf: return vector square root accurate to 2.5ulps.

    This approximation assumes finite math; never returns denormals, but zero;
    does not return the expected values after C89;
    is not compliant with IEEE754 semantics. */

__m128 nr_sqrtvf (__m128 a)
{
  __m128 x0, x1, x2, x3, x4, x5, m, y;

  m = _mm_cmpneq_ps (a, _mm_setzero_ps ()); // m = (a != 0.0? T: F)

  x0 = _mm_rsqrt_ps (a);                    // x0 = initial estimate
  x1 = _mm_and_ps (m, x0);                  // x1 = m & x0
  x2 = _mm_mul_ps (a, x1);                  // x2 = a * x1
  x3 = _mm_mul_ps (_mm_set1_ps (0.5F), x2); // x3 = 0.5 * x2
  x4 = _mm_mul_ps (x1, x2);                 // x4 = x1 * x2
  x5 = _mm_sub_ps (_mm_set1_ps (3.0F), x4); // x5 = 3.0 - x4

  y = _mm_mul_ps (x3, x5);                  // y = x3 * x5
  return (y);                               // y = sqrtf (a)
}
```

These functions return the quotient:

```
#include <xmmintrin.h>

/* nr_divf: return scalar quotient accurate to 1.5ulps.

    This approximation assumes finite math; never returns denormals, but zero;
    does not return the expected values after C89;
    is not compliant with IEEE754 semantics. */

float nr_divf (float a, float b)
{
  __m128 x0, x1, x2, x3;
  float y;

  x0 = _mm_rcp_ss (_mm_set_ss (b));         // x0 = initial estimate
  x1 = _mm_mul_ss (_mm_set_ss (a), x0);     // x1 = a * x0
  x2 = _mm_mul_ss (_mm_set_ss (b), x0);     // x2 = b * x0
  x3 = _mm_sub_ss (_mm_set_ss (2.0F), x2); // x3 = 2 - x2

  _mm_store_ss (&y, _mm_mul_ss (x1, x3));  // y = x1 * x3
  return (y);                               // y = a / b
}

/* nr_divvf: return vector quotient accurate to 1.5ulps.

    This approximation assumes finite math; never returns denormals, but zero;
    does not return the expected values after C89;
    is not compliant with IEEE754 semantics. */

__m128 nr_divf (__m128 a, __m128 b)
{
  __m128 x0, x1, x2, x3, y;

  x0 = _mm_rcp_ps (b);                      // x0 = initial estimate
  x1 = _mm_mul_ps (a, x0);                  // x1 = a * x0
  x2 = _mm_mul_ps (b, x0);                  // x2 = b * x0
  x3 = _mm_sub_ps (_mm_set1_ps (2.0F), x2); // x3 = 2 - x2

  y = _mm_mul_ps (x1, x3);                  // y = x1 * x3
  return (y);                               // y = a / b
}
```

9.8 Use XOR Operations to Negate Operands of SSE*x* Instructions

Optimization

For AMD Family 10h and 12h processors, use instructions that perform XOR operations (XORPS, and XORPD) instead of multiplication instructions to change the sign bits of operands of SSE instructions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

On the AMD Family 10h and 12h processors, using XOR-type instructions allows for more parallelism, since these instructions can execute in either the FADD or FMUL pipe of the floating-point unit. Also, the latency of the MULPS or MULPD instruction is longer than the latency of XORPS or XORPD (see Appendix C, “Instruction Latencies”).

Single Precision

This example shows how to toggle the sign bit of four floating-point values using single-precision SSE instructions:

```
signmask DQ 8000000080000000h,8000000080000000h
xorps xmm0, [signmask] ; Toggle sign bits of all four floats.
```

Double Precision

The following example shows how to toggle the sign bit of two doubles using double-precision SSE instructions:

```
signmask DQ 8000000000000000h,8000000000000000h
xorpd xmm0, [signmask]   ; Toggle sign bit of both doubles.
```

9.9 Clearing MMX™ and XMM Registers with XOR Instructions

Optimization

Use instructions that perform XOR operations (PXOR, XORPS, and XORPD) to clear all the bits in MMX and XMM registers.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The PXOR, XORPS, and XORPD instructions are more efficient than loading a zero value into an MMX or XMM register from memory and then storing it (see Appendix C, “Instruction Latencies,” on page 245). In addition, the processor “knows” that the PXOR, XORPS and XORPD instructions that use the same register for both source and destination do not have a real dependency on the previous contents of the register, and thus, do not have to wait before completing.

Examples

The following examples illustrate how to clear the bits in a register using the different exclusive-OR instructions:

```
; MMX
pxor mm0, mm0       ; Clear the MM0 register.

; SSE
xorps xmm0, xmm0    ; Clear the XMM0 register.

; SSE2
xorpd xmm0, xmm0    ; Clear the XMM0 register.
```

9.10 Finding the Floating-Point Absolute Value of Operands of SSE and SSE2 Instructions

Optimization

Use instructions that perform AND operations (ANDPS, and ANDPD) to determine the absolute value of floating-point operands of SSE and SSE2 instructions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Examples

The following examples illustrate how to clear the sign bits. See Appendix C for latencies of the ANDPS and ANDPD instructions:

```
; SSE
absmask DQ 7FFFFFFF7FFFFFFFh,7FFFFFFF7FFFFFFFh
andps xmm0, [absmask]    ; Clear the sign bits of all four floats in XMM0.
; SSE2
absmask DQ 7FFFFFFFFFFFFFFFh,7FFFFFFFFFFFFFFFh
andpd xmm0, [absmask]    ; Clear the sign bits of both doubles in XMM0.
```

9.11 Accumulating Single-Precision Floating-Point Numbers Using SSE and SSE2 Instructions

Optimization

Careful selection of SSE instructions based on efficient data organization can lead to more economical code.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

SSE and SSE2 provide vectorized multiplication and addition instructions. These instructions are useful for carrying out such operations as complex-number multiplication, 4 ∞ 4 matrix multiplication, and dot products.

Examples

The following example uses SSE instructions. Four floating-point values are loaded into four XMM registers, XMM4–XMM7. These values are then rearranged and added, so as to accumulate the sum of each XMM register into a float in XMM1.

```
;-----------------------------------------------------------------------
; The instructions below take the 4 floats in each XMM register below:
;  xmm4  =  [d,c,b,a]
;  xmm5  =  [D,C,B,A]
;  xmm6  =  [h,g,f,e]
;  xmm7  =  [H,G,F,E]
;
;  and arranges them to look like:
;  xmm4  =  [E,e,A,a]
;  xmm1  =  [F,f,B,b]
;  xmm2  =  [G,g,C,c]
;  xmm3  =  [H,h,D,d]

movaps   xmm3, xmm4   ; xmm3 | [d,c,b,a]
movaps   xmm0, xmm5   ; xmm0 | [D,C,B,A]

unpcklps xmm4, xmm6   ; xmm4 | [f,b,e,a]
unpckhps xmm3, xmm6   ; xmm3 | [h,d,g,c]
movaps   xmm1, xmm4   ; xmm1 | [f,b,e,a]
movaps   xmm2, xmm3   ; xmm2 | [h,d,g,c]

unpcklps xmm5, xmm7   ; xmm5 | [F,B,E,A]
```

```
unpckhps xmm0, xmm7   ; xmm0 | [H,D,G,C]

unpcklps xmm4, xmm5   ; xmm4 | [E,e,A,a]
unpckhps xmm1, xmm5   ; xmm1 | [F,f,B,b]
unpcklps xmm3, xmm0   ; xmm3 | [G,g,C,c]
unpckhps xmm2, xmm0   ; xmm2 | [H,h,D,d]

; Now if we compute the sum of these registers, we get the dot-product
;  of the first row of A with vector X:
;
;  a+b+c+d
;
;  in the lower DWORD of the resultant XMM register. The dot-product of the
;  second row is stored in the second DWORD and so on, such that:
;
;  xmm1 = [V+X+Y+Z,v+x+y+z,A+B+C+D,a+b+c+d]

addps xmm1, xmm4   ; xmm1 | [E+F,e+f,A+B,a+b]
addps xmm3, xmm2   ; xmm3 | [G+H,g+h,C+D,c+d]
addps xmm1, xmm3   ; xmm1 | [E+F+G+H,e+f+g+h,A+B+C+D,a+b+c+d]
```

9.12 Complex-Number Arithmetic Using SSE, SSE2, and SSE3 Instructions

Optimization

Use vectorizing SSE, SSE2 and SSE3 instructions to perform complex number calculations.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Complex numbers have a “real” part and an “imaginary” part (where the imaginary part is denoted by the letter *i*). For example, the complex number *z1* might have a real part equal to 4 and an imaginary part equal to 3, written as 4 + 3i. Multiplying and adding complex numbers is an integral part of digital signal processing. Complex number addition is illustrated here using two complex numbers, *z1* (4 + 3i) and *z2* (5 + 2i):

z1 + *z2* = (4 + 3i) + (5 + 2i) = [4+5] + [3+2]i = 9 + 5i

or:

```
sum.real = z1.real + z2.real
sum.imag = z1.imag + z2.imag
```

Complex number multiplication is illustrated below using the same two complex numbers:

z1 + *z2* = (4 + 3i)(5 + 2i) = [4 ∞ 5 - 3 ∞ 2] + [3 ∞ 5 + 4 ∞ 2]i = 14 + 23i

or:

```
product.real = z1.real * z2.real - z1.imag * z2.imag
product.imag = z1.real * z2.imag + z1.imag * z2.real
```

Complex numbers are stored as streams of two-element vectors, the two elements being the real and imaginary parts of the complex numbers. Addition of complex numbers can be achieved using vectorizing SSE or SSE3 instructions, such as ADDPS and ADDPD. Multiplication of complex numbers is more involved, but SSE3 instructions are available to perform exactly the operations required.

From the formulas for multiplication, the real and imaginary parts of one of the numbers must be interchanged, and, additionally, the products must be positively or negatively accumulated depending upon whether we are computing the imaginary or real portion of the product.

The following functions use SSE, SSE2, and SSE3 instructions to illustrate complex multiplication of streams of complex numbers `x[]` and `y[]` stored in a product stream `prod[]`. For these examples, assume that the sizes of `x[]` and `y[]` are even multiples of four.

Example

Complex Multiplication of Streams of Complex Numbers using SSE3 Instructions

```
; cmplx_multiply_sse3(float *x, float *y, int num_cmplx_elem, float *prod);
;
; TO ASSEMBLE INTO *.obj DO THE FOLLOWING:
; ml64.exe -c cmplx_multiply_sse3.asm
;
;
; define local variable storage offsets
save_xmm6       equ      0h        ;xmmword
save_xmm7       equ      010h      ;xmmword
save_rdi        equ      020h      ;qword
save_rsi        equ      028h      ;qword

stack_size      equ      038h

TEXT   SEGMENT      page      'CODE'
   PUBLIC  cmplx_multiply_sse3
cmplx_multiply_sse3   proc   frame
;===========================================================================
; INSTRUCTIONS BELOW SAVE THE REGISTER STATE WITH WHICH THIS ROUTINE WAS ENTERED
; REGISTERS RSI, ESI, and XMM6, ARE CONSIDERED VOLATILE AND ASSUMED TO BE CHANGED
; WHILE THE REGISTERS BELOW MUST BE PRESERVED IF THE USER IS CHANGING THEM
;
; NOTE the use of the Masm 64 prolog macros needed for Structured Exception
; Handling.
   sub      rsp,stack_size
```

```
   .ALLOCSTACK   stack_size
   movdqa   XMMWORD PTR [rsp+save_xmm6],xmm6   ; save xmm6
   .SAVEXMM128 xmm6, save_xmm6
   movdqa   XMMWORD PTR [rsp+save_xmm7],xmm7   ; save xmm7
   .SAVEXMM128 xmm7, save_xmm7
   mov      QWORD PTR [rsp+save_rdi],rdi       ; save rdi
   .SAVEREG rdi, save_rdi
   mov      QWORD PTR [rsp+save_rsi],rsi       ; save rsi
   .SAVEREG rsi, save_rsi
   .ENDPROLOG
;============================================================================
; Parameters passed into routine according to the Microsoft AMD64 ABI:
; rcx = ->x
; rdx = ->y
; r8d = num_cmplx_elem
; r9 = ->prod
;============================================================================

mov rsi, rcx    ; rsi = ->x
mov rdi, rdx    ; rdi = ->y
mov ecx, r8d    ; rcx = num_cmplx_elem (zero extends the 64 bit destination
                ; register)

;============================================================================
; THE 6 ASM LINES BELOW OFFSET THE ADDRESS TO THE ARRAYS x[] AND y[] SUCH
; THAT THEY CAN BE ACCESSED IN THE MOST EFFICIENT MANNER AS ILLUSTRATED
; BELOW IN THE LOOP mult8cmplxnum_loop WITH THE MINIMUM NUMBER OF
; ADDRESS INCREMENTS
;============================================================================
mov r8, rcx     ; rdx = num_cmplx_elem
neg rcx         ; rcx = -num_cmplx_elem
imul r8, 8      ; edx = 8 * num_cmplx_elem = # bytes in x[] and y[] to multiply
add rsi, r8     ; esi = -> to last element of x[] to multiply
add rdi, r8     ; edi = -> to last element of y[] to multiply
add r9, r8      ; r9 = -> end of prod[] to calculate
;============================================================================
; THIS LOOP MULTIPLIES 8 COMPLEX #s FROM "x[]" UPON 8 COMPLEX #s FROM "y[]"
; AND RETURNS THE PRODUCT IN "prod[]".
;============================================================================
ALIGN 32 ; Align address of loop to a 32-byte boundary.
four_cmplx_prod_loop: ;
movaps xmm0, XMMWORD PTR [rsi+rcx*8]     ; xmm0=[x1i,x1r,x0i,x0r]
movaps xmm1, XMMWORD PTR [rsi+rcx*8+16]  ; xmm1=[x3i,x3r,x2i,x2r]

movaps xmm4, XMMWORD PTR [rdi+rcx*8]     ; xmm4=[y1i,y1r,y0i,y0r]
movaps xmm5, XMMWORD PTR [rdi+rcx*8+16]  ; xmm5=[y3i,y3r,y2i,y2r]

movaps    xmm6,xmm4           ; xmm6=[y1i,y1r,y0i,y0r]
movaps    xmm7,xmm5           ; xmm7=[y3i,y3r,y2i,y2r]
movshdup xmm2,xmm0            ; xmm2=[x1i,x1i,x0i,x0i]
movshdup xmm3,xmm1            ; xmm3=[x3i,x3i,x2i,x2i]
movsldup xmm0,xmm0            ; xmm0=[x1r,x1r,x0r,x0r]
movsldup xmm1,xmm1            ; xmm1=[x3r,x3r,x2r,x2r]
shufps   xmm6,xmm4,0b1h       ; xmm6=[y1r,y1i,y0r,y0i]
shufps   xmm7,xmm5,0b1h       ; xmm7=[y3r,y3i,y2r,y2i]
```

```
mulps xmm2, xmm6              ; xmm2=[x1i*y1r,x1i*y1i,x0i*y0r,x0i*y0i]
mulps xmm3, xmm7              ; xmm3=[x3i*y3r,x3i*y3i,x2i*y2r,x2i*y2i]
mulps xmm0, xmm4              ; xmm0=[x1r*y1i,x1r*y1r,x0r*y0i,x0r*y0r]
mulps xmm1, xmm5              ; xmm1=[x3r*y3i,x3r*y3r,x2r*y2i,x2r*y2r]

addsubps xmm0, xmm2 ; xmm0=[x1r*y1i+x1i*y1r,x1r*y1r-x1i*y1i,
                    ; x0r*y0i+x0i*y0r,x0r*y0r-x0i*y0i]

addsubps xmm1, xmm3 ; xmm1=[x3r*y3i+x3i*y3r,x3r*y3r-x3i*y3i,
                    ;       x2r*y2i+x2i*y2r,x2r*y2r-x2i*y2i]

movntps XMMWORD PTR [r9+rcx*8], xmm0   ; Stream XMM0-XMM3 to representative memory
movntps XMMWORD PTR [r9+rcx*8+16], xmm1
add rcx, 4 ; RCX = RCX +4
jnz four_cmplx_prod_loop
sfence                          ; Finish all memory writes.

;===========================================================================
; INSTRUCTIONS BELOW RESTORE THE REGISTER STATE WITH WHICH THIS ROUTINE WAS
; ENTERED
; REGISTERS RDI, RSI ARE CONSIDERED VOLATILE AND ASSUMED TO BE CHANGED
; WHILE THE REGISTERS BELOW MUST BE PRESERVED IF THE USER IS CHANGING THEM
   mov   rdi, QWORD PTR [rsp+save_rdi]      ; restore rdi
   mov   rsi, QWORD PTR [rsp+save_rsi]      ; restore rsi
   movdqa   xmm6, [rsp+save_xmm6]           ; restore xmm6
   movdqa   xmm7, [rsp+save_xmm7]           ; restore xmm7
   add   rsp,stack_size
   ret

cmplx_multiply_sse3      endp
TEXT   ENDS
END
```

The example above makes use of many optimization techniques. First, the SSE3 technology code utilizes the MOVSHDUP, MOVSLDUP and ADDSUBPS instructions, whose operations are outlined below:

```
; MOVSHDUP
; Suppose that XMM0 contains four floats: r2, i2, r1 and i1.
; INPUT:
; XMM0 = [i2,r2,i1,r1]
; OUTPUT:
; XMM1 = [i2,i2,i1,i1]
movshdup xmm1, xmm0 ; SMM1 = [i2,i2,i1,i1]

; MOVSLDUP
; Suppose that XMM0 contains four floats: r2, i2, r1 and i1.
; INPUT:
; XMM0 = [i2,r2,i1,r1]
; OUTPUT:
; XMM1 = [r2,r2,r1,r1]
movsldup xmm1, xmm0 ; XMM1 = [r2,r2,r1,r1]
```

The ADDSUBPS instruction is specifically designed for use in complex arithmetic operations.

```
; ADDSUBPS
; Suppose that XMM0 contains four floats: r3 * i4, r3 * r4 , r1 * i2, r1 * r2
; where r3 * i4 and r1 * i2 are the products of the real
; part of the first complex number and the imaginary part of the second
; complex number for two pairs of complex numbers,
; and  r3 * r4 and r1 * r2 are the product of the real parts
; of two pairs of complex numbers
; Also suppose that XMM1 contains four floats: i3 * r4, i3 * i4, i1 * r2, i1 * i2
; where i1 * r2  and i3 * r4  are the products of the imaginary part of the
; first complex number and the real part of the second complex number
; for two pairs of complex numbers
; and i3 * i4 and i1 * i2 air the products of the imaginary parts
; of two pairs of complex numbers.
; INPUTS:
; XMM0 = [r3*i4,r3*r4,r1*i2,r1*r2]
; XMM1 = [i3*r4,i3*i4,i1*r2,i1*i2]
; OUTPUT:
; XMM0 = [r3*i4+i3*r4+,r3*r4-i3*i4,r1*i2+i1*r2,r1*r2-i1*i2]
addsubps xmm0, xmm1 ; XMM0 = [r3*i4+i3*r4+,r3*r4-i3*i4,r1*i2+i1*r2,r1*r2-i1*i2]
```

The second optimization is a form of loop unrolling so that four complex numbers are concurrently multiplied in the examples using SSE and SSE3 instructions to break up register dependencies. Loads, multiplications, and additions do not execute with zero delay, but have a latency associated with them. The following instructions are interdependent:

```
movaps xmm0, XMMWORD PTR [rsi+rcx*8]     ; xmm0=[x1i,x1r,x0i,x0r]
movaps xmm4, XMMWORD PTR [rdi+rcx*8]     ; xmm4=[y1i,y1r,y0i,y0r]

movaps    xmm6,xmm4           ; xmm6=[y1i,y1r,y0i,y0r]
movshdup xmm2,xmm0            ; xmm2=[x1i,x1i,x0i,x0i]
movsldup xmm0,xmm0            ; xmm0=[x1r,x1r,x0r,x0r]
shufps   xmm6,xmm4,0b2h       ; xmm6=[y1r,y1i,y0r,y0i]
mulps xmm2, xmm6              ; xmm2=[x1i*y1r,x1i*y1i,x0i*y0r,x0i*y0i]
mulps xmm0, xmm4              ; xmm0=[x1r*y1i,x1r*y1r,x0r*y0i,x0r*y0r]

addsubps xmm0, xmm2           ; xmm0=[x1r*y1i+x1i*y1r,x1r*y1r-x1i*y1i,
                              ;       x0r*y0i+x0i*y0r,x0r*y0r-x0i*y0i]
```

The move from memory (MOVAPS) requires 2 cycles (assuming that the data is available in L1 cache), MOVSHDUP, MOVSLDUP require 2 cycles each, the two MULPS instructions require 4 cycles, the SHUFPS requires 4 cycles, and ADDSUBPS requires 4 cycles. The instruction flow through the processor is illustrated on a clock-cycle basis, as follows:

```
Instruction 0     2     4     6     8     10    12    14    16
MOVAPS      xxxxxx
MOVAPS      xxxxxx
MOVAPS            xxxxxx
MOVSHDUP          xxxxxx
MOVSLDUP          xxxxxx
SHUFPS                  xxxxxxxxxxxx
MULPS                             xxxxxxxxxxxx
```

```
MULPS                    xxxxxxxxxxxx
ADDSUBPS                                          xxxxxxxxxxxx
```

These two complex multiplies take 15 cycles to finish. During these 15 cycles, the processor has the ability to perform 60 single-precision adds and 60 single-precision multiplies, but in this code sequence it only performs eight multiplies and four adds (the subtracts are performed on the ADD execution unit). This is only 10% utilization. The majority of the time is spent waiting for previous instructions to terminate so that arguments to future instructions are available. By unrolling the multiplication and working with four complex numbers per loop, there are more instructions that are not dependent on previous or presently executing operations. This allows the processor to mask the execution latency and keep itself busier, as illustrated below:

```
Instruction 0     2     4     6     8     10    12    14    16    18
MOVAPS      xxxxxx
MOVAPS      xxxxxx
MOVAPS         xxxxxx
MOVAPS         xxxxxx
MOVAPS            xxxxxx
MOVAPS            xxxxxx
MOVSHDUP          xxxxxx
MOVSHDUP             xxxxxx
MOVSLDUP             xxxxxx
MOVSLDUP             xxxxxx
SHUFPS                  xxxxxxxxxxxx
SHUFPS                  xxxxxxxxxxxx
MULPS                             xxxxxxxxxxxx
MULPS                        xxxxxxxxxxxx
MULPS                                xxxxxxxxxxxx
MULPS                           xxxxxxxxxxxx
ADDSUBPS                                          xxxxxxxxxxxx
ADDSUBPS                                             xxxxxxxxxxxx
```

Multiplying four complex single-precision numbers only takes 17 cycles as opposed to 15 cycles to multiply two complex single-precision numbers. The floating-point pipes are kept busier by feeding new instructions into the floating-point pipeline each cycle. In the arrangement above, 16 multiplies and 8 additions are performed in 17 cycles, achieving a 1.8x increase in performance. Unrolling the loop one more time will improve efficiency even more, at the expense of requiring all 16 XMM registers at once.

The last optimization uses MOVNTPS instructions—nontemporal writes to memory that stream data to main memory. These instructions increase throughput to memory and make more efficient use of the bandwidth provided by the processor and memory controller. Nontemporal writes, such as MOVNTPS, and MOVNTDQ, should only be used on data that is not going to be accessed again in the near future.

9.13 Optimized 4 X 4 Matrix Multiplication on 4 X 1 Column Vector Routines

Optimization

Transpose the rotation matrix to eliminate the need to accumulate floating-point values in an XMM register.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The multiplication of a 4 × 4 matrix with a 4 × 1 vector is commonly used in 3-D graphics for geometric transformation (translating, scaling, rotating, and applying perspective to 3-D points represented in homogeneous coordinates). Efficiency in single-precision matrix multiplication can be enhanced by use of SIMD instructions to increase throughput, but there are other general optimizations that can be implemented to further increase performance. The first optimization is the transposition of the rotation matrix such that column *n* of the matrix becomes row *n* and row *m* becomes column *m*. There are no SSE or SSE2 instructions that accumulate the floats and doubles in a single XMM register; for this reason, the matrix must be transposed. If the rotation matrix is not transposed, then the dot-product of a row of the matrix with a column vector necessitates the accumulation of the four floating-point values in an XMM register. The multiplication on the column vector is illustrated here

```
                |r00 r01 r02 r03|         |r00 r10 r20 r30|   |v0|   |v'0|
tr(R) × v = tr  |r10 r11 r12 r13| × v =   |r01 r11 r21 r31| × |v1| = |v'1|
                |r20 r21 r22 r23|         |r02 r12 r22 r32|   |v2|   |v'2|
                |r30 r31 r32 r33|         |r03 r13 r23 r33|   |v3|   |v'3|

          Step 0        Step 1        Step 2        Step 3
|v'0|    |r00 × v0|    |r01 × v1| + |r02 × v2| + |r03 × v3|
|v'1| =  |r10 × v0| +  |r11 × v1| + |r12 × v2| + |r13 × v3|
|v'2|    |r20 × v0|    |r21 × v1| + |r22 × v2| + |r23 × v3|
|v'3|    |r30 × v0|    |r31 × v1| + |r32 × v2| + |r33 × v3|
```

In each step above, the elements of the rotation matrix can be loaded into an XMM register with the MOVAPS instruction, assuming the rotation matrix begins at a 16-byte-aligned memory location. Transposition of the rotation matrix eliminates the need to accumulate the floating-point values in an XMM register, but it does require the duplication of the elements of the 4 × 1 column vector V in all four floating-point values of the XMM register in each step above. The following example shows an

SSE function that performs 4 × 4 matrix multiplication upon a stream of num_vertices_to_rotate vertices.

Example

4 X 4 Matrix Multiplication (SSE)

```
; matrix_x_vector_sse(float *trR, float *v, int num_vertices_to_rotate,
float *rotv);
;
; TO ASSEMBLE INTO *.obj DO THE FOLLOWING:
; ml.exe -coff -c matrix_x_vector_sse.asm
;
.586
.K3D
.XMM
_TEXT SEGMENT
PUBLIC _matrix_x_vector_sse
_matrix_x_vector_sse PROC NEAR
;=============================================================================
; INSTRUCTIONS BELOW SAVE THE REGISTER STATE WITH WHICH THIS ROUTINE WAS
; ENTERED.
; REGISTERS EAX, ECX, AND EDX ARE CONSIDERED VOLATILE AND ASSUMED TO BE CHANGED,
; WHILE THE REGISTERS BELOW MUST BE PRESERVED IF THE USER IS CHANGING THEM
push ebp
mov ebp, esp
;=============================================================================
; Parameters passed into routine:
; [ebp+8] = ->trR
; [ebp+12] = ->v
; [ebp+16] = num_vertices_to_rotate
; [ebp+20] = ->rotv
;=============================================================================
push ebx
push esi
push edi
;=============================================================================
; THE 4 ASM LINES BELOW LOAD THE FUNCTION's ARGUMENTS INTO GENERAL-PURPOSE
; REGISTERS (GPRS)
; esi = address of Transposed Rotation Matrix
; edi = address of vertices to rotate
; ecx = # of vertices to rotate
; eax = address of rotated vertices
;=============================================================================
mov esi, [ebp+8] ; ESI = ->trR
mov edi, [ebp+12] ; EDI = ->v
mov ecx, [ebp+16] ; ECX = num_vertices_to_rotate
mov edx, ecx ; EDX = num_vertices_to_rotate
shl edx, 4 ; EDX = 16*num_vertices_to_rotate
mov eax, [ebp+20] ; EAX = ->rotv
imul ecx, 2 ; ECX = # quadwords of vertices to rotate
add edi, edx ; EDI = -> end of "v"
add eax, edx ; EAX = -> end of "rotv"
neg ecx ; ECX = -# quadwords of vertices to rotate
;=============================================================================
```

```
; THE 4 ASM LINES BELOW LOAD THE TRANSPOSED ROTATION MATRIX "R" INTO XMM0-XMM3
; IN THE FOLLOWING MANNER:
; xmm0 = column 0 of "R" or row 0 of "R" transpose
; xmm1 = column 1 of "R" or row 1 of "R" transpose
; xmm2 = column 2 of "R" or row 2 of "R" transpose
; xmm3 = column 3 of "R" or row 3 of "R" transpose
;=============================================================================
movaps xmm0, [esi] ; XMM0 = [R30,R20,R10,R00]
movaps xmm1, [esi+16] ; XMM1 = [R31,R21,R11,R01]
movaps xmm2, [esi+32] ; XMM2 = [R32,R22,R12,R02]
movaps xmm3, [esi+48] ; XMM3 = [R33,R23,R13,R03]
;=============================================================================
; THIS LOOP ROTATES "num_vertices_to_rotate" VERTICES BY THE TRANSPOSED
; ROTATION MATRIX "R" PASSED INTO THE ROUTINE AND STORES THE ROTATED
; VERTICES TO "rotv".
;=============================================================================
ALIGN 32 ; Align address of loop to a 32-byte boundary.
rotate_vertices_loop:
movlps xmm4, [edi+8*ecx] ; XMM4=[,,v1,v0]
movlps xmm6, [edi+8*ecx+8] ; XMM6=[,,v3,v2]
unpcklps xmm4, xmm4 ; XMM4=[v1,v1,v0,v0]
unpcklps xmm6, xmm6 ; XMM6=[v3,v3,v2,v2]
movhlps xmm5, xmm4 ; XMM5=[,,v1,v1]
movhlps xmm7, xmm6 ; XMM7=[,,v3,v3]
movlhps xmm4, xmm4 ; XMM4=[v0,v0,v0,v0]
mulps xmm4, xmm0 ; XMM4=[R30*v0,R20*v0,R10*v0,R00*v0]
movlhps xmm5, xmm5 ; XMM5=[v1,v1,v1,v1]
mulps xmm5, xmm1 ; XMM5=[R31*v1,R21*v1,R11*v1,R01*v1]
movlhps xmm6, xmm6 ; XMM6=[v2,v2,v2,v2]
mulps xmm6, xmm2 ; XMM6=[R32*v2,R22*v2,R12*v2,R02*v2]
addps xmm4, xmm5 ; XMM4=[R30*v0+R31*v1,R20*v0+R21*v1,
; R10*v0+R11*v1,R00*v0+R01*v1]
movlhps xmm7, xmm7 ; XMM7=[v3,v3,v3,v3]
mulps xmm7, xmm3 ; XMM6=[R33*v3,R23*v3,R13*v3,R03*v3]
addps xmm6, xmm7 ; XMM6=[R32*v2+R33*v3,R22*v2+R23*v3,
; R12*v2+R13*v3,R02*v2+R03*v3]
addps xmm4, xmm6 ; XMM4=New rotated vertex
movntps [eax+8*ecx], xmm4 ; Store rotated vertex to rotv.
add ecx, 2 ; Decrement the # of QWORDs to rotate by 2.
jnz rotate_vertices_loop
sfence ; Finish all memory writes.
;=============================================================================
; INSTRUCTIONS BELOW RESTORE THE REGISTER STATE WITH WHICH THIS ROUTINE
; WAS ENTERED
; REGISTERS EAX, ECX, EDX ARE CONSIDERED VOLATILE AND ASSUMED TO BE CHANGED
; WHILE THE REGISTERS BELOW MUST BE PRESERVED IF THE USER IS CHANGING THEM
pop edi
pop esi
pop ebx
mov esp, ebp
pop ebp
;=============================================================================
ret
_matrix_x_vector_sse ENDP
_TEXT ENDS
END
```

To greatly enhance performance, the previous function can perform the matrix multiplication not only on one four-column vector, but on many. Creating a separate function to transform a single vertex and repeatedly calling the function is prohibitively expensive because of the overhead in pushing and popping registers from the stack. This applies to routines that negate a single vector, nullify a single vector, and add two vectors.

9.14 Floating-Point-to-Integer Conversion

Optimization

Floating-point-to-integer conversion in C and C++ requires the use of truncation. Use one of the instructions from CVTTSS2SI, CVTTSD2SI to convert a floating-point number to integer when truncation is required. See the *AMD64 Architecture Programmer's, Volume 4: 128-Bit Media Instructions*, order# 26568, for details.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

These instructions provide the fastest means by which to convert floating-point types to integers in AMD Family 10h and 12h processors.

9.15 Reuse of Dead Registers

Optimization

On AMD Family 10h and 12h processors, when it is necessary to save the contents of a register that is a single-precision floating-point scalar to another unused (or *dead*) register, use `MOVAPS dest, src` instead of `MOVSS dest, src`.

When saving a register that is a double-precision floating-point scalar to another register, where the contents are unknown, then use `MOVAPD dest, src` instead of `MOVSD dest, src`.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

On the AMD Family 10h and 12h processors, the `MOVSS dest, src` instruction takes additional time to execute if any of the upper three fields of `dest` is a denormal. Additionally, the `MOVSS dest, src` instruction has a dependency on previous instructions that change `dest`, either partially or in full, and the `MOVAPS dest, src` instruction breaks such dependency chains by changing `dest` as a whole.

The `MOVSD dest, src` instruction also takes additional time to execute, if the previous value in `xmm1` is a denormal. Moreover, the `MOVSD dest, src` instruction has a dependency on previous instructions that change `dest` either partially or in full. On the other hand, the `MOVAPD dest, src` instruction breaks such dependency chains by writing to all of `dest`.

9.16 Floating-Point Scalar Conversions

Optimization

Use the recommended instruction sequences given in Table 9 and Table 10 to convert integer data to floating-point data.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

On AMD Family 10h and 12h processors, some SIMD conversion instructions are VectorPath and/or add a false dependency on previous instructions that change the same destination register. In the cases for which there are alternatives in Tables 9 and 10, these instruction sequences use DirectPath instructions and provide better performance. (All recommendations apply to both 32-bit and 64-bit software, unless stated otherwise.)

Several instructions may be required to perform some conversions from unsigned integer to floating-point, due to the lack of a suitable conversion instruction, therefore signed integers should be favored when converting to floating-point.

Table 9. Single-Precision Floating-Point Scalar Conversion

Conversion	From a Register	From Memory
32-Bit Signed Integer to Single-Precision	`movd xmm, reg32` `cvtdq2ps xmm, xmm`	`movd xmm, mem32` `cvtdq2ps xmm, xmm`

Table 9. Single-Precision Floating-Point Scalar Conversion

Conversion	From a Register	From Memory
32-Bit Unsigned Integer to Single-Precision	64-bit software: `xorps xmm, xmm`[2] `mov mem64, reg64` `cvtsi2ss xmm, mem64`	64-bit software: `xorps xmm, xmm`[2] `cvtsi2ss xmm, mem64`
64-Bit Signed Integer to Single-Precision	64-bit software: `xorps xmm, xmm`[2] `mov mem64, reg64` `cvtsi2ss xmm, mem64`	64-bit software: `xorps xmm, xmm`[2] `cvtsi2ss xmm, mem64`
Double-Precision to Single-Precision	`unpcklpd xmm2, xmm2`[1] `cvtpd2ps xmm1, xmm2`	`movsd xmm, mem64` `cvtpd2ps xmm, xmm`

Notes:

1. *If the contents of [127:64] of* xmm2 *is known to be a normal number, this instruction can be omitted.*
2. *This avoids a merge dependency for contents of [127:32] of xmm as a result of a previous long latency instruction that has written to contents of [127:0] of xmm.*

Table 10. Double-Precision Floating-Point Scalar Conversion

Conversion	From a Register	From Memory
32-Bit Signed Integer to Double-Precision	`movd xmm, reg32` `cvtdq2pd xmm, xmm`	`movd xmm, mem32` `cvtdq2pd xmm, xmm`
32-Bit Unsigned Integer toDouble-Precision	64-bit software: `xorpd xmm, xmm`[2] `mov mem64, reg64` `cvtsi2sd xmm, mem64`	64-bit software: `xorpd xmm, xmm`[2] `cvtsi2sd xmm, mem64`
64-Bit Signed Integer to Double-Precision	64-bit software: `xorpd xmm, xmm`[2] `mov mem64, reg64` `cvtsi2sd xmm, mem64`	64-bit software: `xorpd xmm, xmm`[2] `cvtsi2sd xmm, mem64`
Single-Precision to Double-Precision	`unpcklps xmm2, xmm2`[1] `cvtps2pd xmm1, xmm2`	`movss xmm, mem32` `cvtps2pd xmm, xmm`

Notes:

1. *If the contents of [63:32] of xmm2 is known to be a normal number, this instruction can be omitted.*
2. *This avoids a merge dependency for contents of [127:32] of xmm as a result of a previous long latency instruction that has written to contents of [127:0] of xmm.*

9.17 Move/Compute Optimization

Optimization

The latency of certain XMM move instructions that provide an input operand to a subsequent compute instruction can be hidden in many cases. The hardware will only recognize such optimization opportunities relative to the most recent move instruction, so instructions must be ordered with this in mind. Use these specific move instructions, appropriately ordered as described below, wherever possible.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Carefully ordering move instructions with respect to subsequent computations having a source XMM that is identical to the destination XMM of the move increases throughput by effectively canceling the latency cost of move operations associated with FPU computations.

This hardware optimization is designed to work with the MOVAPD, MOVAPS, MOVDQA, MOVDQU, MOVUPD, and MOVUPS instructions. Other SSE move instructions cause a two- cycle delay in executing the dependent compute instruction. If at all possible, every effort should be made to use move instructions that the processor hardware can optimize.

Move-Compute Execution Rules

Move instructions themselves cannot short-circuit the latency of a prior move instruction on which they are dependent; this can only be done by compute instructions.

Only one move instruction's output-to-input mapping can be recognized at a time and eliminated for dependent compute instructions. So a given move-compute optimization opportunity exists only until one of the following occurs:

- The destination or source XMM of the currently-recognized move instruction is overwritten by a younger instruction.
- Another eligible move instruction is encountered.
- A redirect occurs (such as from a mispredicted branch).

Example 1

Avoid move instructions that the move-compute hardware does not recognize:

Original sequence—move instructions aren't recognized by move-compute hardware:

```
movsd  xmm2, xmm1
mulsd  xmm2, xmm2; mulsd can't start until movsd completes
movsd  xmm4, xmm3
mulsd  xmm4, xmm4; mulsd can't start until movsd completes
```

Optimal sequence—move instructions are recognized as move-compute eligible:

```
movapd xmm2, xmm1; hardware recognizes xmm2 = xmm1
mulsd  xmm2, xmm2; executed as xmm2 = xmm1 * xmm1
movapd xmm4, xmm3; hardware recognizes xmm4 = xmm3
mulsd  xmm4, xmm4; executed as xmm4 = xmm3 * xmm3
```

Less optimal sequence due to poor instruction ordering:

```
movapd xmm2, xmm1; hardware recognizes xmm2 = xmm1
movapd xmm4, xmm3; hardware loses sight of xmm2 = xmm1
mulsd  xmm2, xmm2; not optimized by hardware -- stalls on movapd result
mulsd  xmm4, xmm4; executed as xmm4 = xmm3 * xmm3 -- no stall
```

Non-interacting instructions, including ineligible move instructions, may appear anywhere in these sequences without affecting the hardware optimization opportunities, although the biggest gain occurs when a dependent compute instruction can issue at the same time as the move.

Example 2

In this example, an intervening SUBPD instruction does not affect the destination register of the previous MOVAPD instruction, which the optimization is able to cut out of the critical path for the subsequent ADDPD instruction.

```
addpd xmm2, xmm1
movapd xmm3, xmm2 ; Hardware recognizes xmm3 = xmm2.
subpd xmm4, xmm0  ; Optimization is not applicable, since neither source operand
                  ; matches the  previous move's destination.
addpd xmm4, xmm3  ; Executes as xmm4 = xmm4 + xmm2 to cut MOVAPD
                  ; out of critical path.
addpd xmm3, xmm2  ; Executes as xmm3 = xmm2 + xmm2, then cancels further use of
                  ; this optimization because xmm3 is overwritten
```

Example 3

Here, the interfering instruction blocks the hardware optimization opportunity.

```
addpd xmm2, xmm1
movapd xmm3, xmm2 ; Hardware recognizes that xmm3 = xmm2.
subpd xmm2, xmm0  ; xmm3 no longer equal to xmm2, hence move-compute
                  ; optimization not possible
addpd xmm4, xmm3  ; Optimization not enabled due to intervening SUBDP xmm2, xmm0
```

9.18 Scalar Merge Optimization

Optimization

Using Scalar Merge Optimization with AMD Family 12h processors, the SIMD conversion instructions that have the operand from memory and zero out the higher 64-bit of the 128-bit XMM register, do not induce false dependencies on subsequent instructions.

Rationale

On Family 10h processors, instructions that zero-out the higher 64-bits of XMM registers introduce false dependency on subsequent instructions that only consume the lower 64-bit of the XMM registers. This optimization removes this false-dependency and in-effect the instruction does not have to wait for its merge dependency to resolve before it can be scheduled for execution.

Example 1

Instructions that zero-out the higher 64-bits are as follows:

```
MOVD xmm, mem64
MOVQ xmm, mem
MOVQ2DQ xmm, mmx
MOVSD xmm, mem
CVTPD2PS xmm, mem
CVTPD2DQ xmm, mem
CVTTPD2DQ xmm,mem
MOVD xmm, reg32/mem32
MOVSS xmm, mem
```

Following are dependent instructions which can potentially benefit from the optimization:

```
CVTPI2PS xmm, mem
CVTSI2SD xmm, mem
CVTSS2SD xmm1,mem
MOVHLPS xmm1, xmm2
MOVLPD xmm, mem
MOVLPS xmm, mem
MOVSD xmm1, xmm2
SQRTSD xmm1, mem

CVTSD2SS xmm1, mem
CVTSI2SS xmm, mem
MOVSS xmm1, xmm2
RSQRTSS xmm1, mem
SQRTSS xmm1, mem
```

Chapter 10 x87 Floating-Point Optimizations

AMD Family 10h and 12h processors support multiple methods of performing floating-point operations, including the older x87 assembly instructions, in addition to the more recent SIMD instructions (SSE, SSE2, SSE3, and SSE4a technologies).

AMD Family 10h and 12h processors are 64-bit processors that are fully backwards compatible with 32-bit code. In general, 64-bit operating systems support the x87 instructions in 32-bit threads; however, 64-bit operating systems may not support x87 instructions in 64-bit threads. To facilitate future migration from 32-bit to 64-bit code, it may be necessary to avoid x87 instructions altogether and use only SSE, SSE2, SSE3, and SSE4a instructions when writing new 32-bit code.

Note that x87 and scalar SSE instructions cannot schedule floating-point operations in the new second set of FPU registers. Thus, the x87 instructions cannot take full advantage of the new 128-bit floating-point resources; better performance is achieved using packed SSE instructions.

This chapter details the methods used to optimize floating-point code to the pipelined x87 floating-point registers.

This chapter covers the following topics:

Topic	Page
Using Multiplication Rather Than Division	173
Achieving Two Floating-Point Operations per Clock Cycle	174
Floating-Point Compare Instructions	178
Using the FXCH Instruction Rather Than FST/FLD Pairs	178
Floating-Point Subexpression Elimination	179
Accumulating Precision-Sensitive Quantities in x87 Registers	180
Avoiding Extended-Precision Data	181

10.1 Using Multiplication Rather Than Division

Optimization

If accuracy requirements allow, convert floating-point division by a constant to multiplication by the reciprocal.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Divisors that are powers of two—and their reciprocals—are exactly representable, and therefore do not cause an accuracy issue, except for the rare case in which the reciprocal overflows or underflows. Unless such an overflow or underflow occurs, always convert a division by a power of two for multiplication. (Underflow in a reciprocal operation can only occur if one is flushing denormals to zero.) Although AMD Family 10h and 12h processors have high-performance division, multiplication is significantly faster than division. (This method will likely be faster than using the FSCALE instruction, and considerably faster than many implementations of the **scalb()** function for x87.

10.2 Achieving Two Floating-Point Operations per Clock Cycle

Optimization

Pay special attention to the order and packing of the operations to sustain up to two floating-point operations per clock cycle.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The AMD Family 10h and 12h floating-point unit can sustain up to two floating-point operations (one in the add unit and one in the multiply unit) per clock cycle. However, to achieve this, you must pay special attention to the order and packing of the operations. For example, consider multiplying a 30 ∞ 30 double-precision matrix **A** by a transposed 30 ∞ 30 double-precision matrix **B**, the result of which is called **C**.

Use Efficient Addressing of FPU Data When Loading and Storing

The rows of **A** are 240 bytes wide, as are the columns of **B**. There are eight x87 floating-point registers [ST(0)–ST(7)], and in this example, six rows of **A** are concurrently multiplied by a single column of **B**. The address of the first element of the first row of **A** (A[0]) is presumed to be stored in the EDI register, while the address of the first element of the first column of **B** (B[0]) is stored in ESI.

This addressing scheme initially appears to be a good idea, however, only 128 bytes can be addressed forward of A[0] with 8-bit offsets (hence 3-byte instructions are required—two bytes for the instruction and one byte for the offset). When the offset is greater than 128 bytes from the address in

the general-purpose register, the size of the instruction increases from 3 bytes to 6 bytes (offsets larger than 128 bytes are represented by 32 bits rather than 8 bits in the instruction). Address offsets larger than 128 bytes require 6-byte instructions, as these offsets require 32 bits rather than 8 bits in the instruction. Large instruction sizes reduce the number of decoded operations to be executed within the pipes of the floating-point unit, and as such prevent us from achieving two floating-point operations per clock cycle. To alleviate this, the general-purpose registers EDI and ESI are offset by 128 bytes such that they contain the addresses of A[15] and B[15]. This is beneficial because data within 128 bytes (16 double-precision numbers) before or after these two locations can now be accessed with instructions that are 2–3 bytes in size. The next five rows of **A** can be efficiently addressed in terms of the first row. Storing the size of a single row of **A** (240 bytes) in the EAX register, the size of three rows (720 bytes) in EBX, and the size of five rows (1200 bytes) in ECX, the first element of rows 0–5 of **A** can be addressed as follows:

```
fld QWORD PTR [edi-128]          ; Load A[i,0].
fld QWORD PTR [edi+eax-128]      ; Load A[i+1,0].
fld QWORD PTR [edi+eax*2-128]    ; Load A[i+2,0].
fld QWORD PTR [edi+ebx-128]      ; Load A[i+3,0].
fld QWORD PTR [edi+eax*4-128]    ; Load A[i+4,0].
fld QWORD PTR [edi+ecx-128]      ; Load A[i+5,0].
```

This addressing scheme reduces the size of all loads from memory to 3 bytes; additionally, to address rows 6–11 of **A**, you only need to add 240*6 to EDI.

Avoid Register Dependencies by Spacing Apart Instructions that Accumulate Results in a Register

The second general optimization to consider is spacing out register dependencies. Operations internally in the floating-point unit have an execution latency (normally 3–4 cycles for x87 operations). Consider this instruction sequence:

```
fldz                         ; Push 0.0 onto floating-point stack.
fld  QWORD PTR [edi-128]     ; Push A[i,0] onto stack.
fmul QWORD PTR [esi-128]     ; Multiply A[i,0] by B[0,j].
faddp st(1), st(0)           ; Accumulate contribution to dot product of
                             ; A's row i and B's column j.
fld  QWORD PTR [edi-120]     ; Push A[i,1] onto stack.
fmul QWORD PTR [esi-120]     ; Multiply A[i,1] by B[1,j].
faddp st(1), st(0)           ; Accumulate contribution to dot product of
                             ; A's row i and B's column j.
fld  QWORD PTR [edi-112]     ; Push A[i,2] onto stack.
fmul QWORD PTR [esi-112]     ; Multiply A[i,2] by B[2,j].
faddp st(1), st(0)           ; Accumulate contribution to dot product of
                             ; A's row i and B's column j.
```

The second statement loads A[0] into ST(0), and the third statement multiplies it by B[0]. The subsequent line adds this product to ST(1), where the dot product of row 0 of matrix **A** and column 0 of matrix **B** is accumulated. Each of the subsequent groups of three instructions adds the contribution of the remaining 29 elements to the dot product. This code is poor because all the addition operations depend upon the contents of a single register, ST(1). The AMD Family 10h and 12h processors have out-of-order-execution floating-point units, but none of the addition operations can be performed out

of order because the result of each addition operation depends on the outcome of the previous addition operation. Instruction scheduling based on this code greatly limits the throughput of the floating-point unit. To alleviate this, space out operations that are dependent on one another. In this case, work with six rows of **A** rather than one at a time, as follows:

```
; Multiply first element of each of six rows of A by first element of
;  B's column j.
fldz                        ; Push 0.0 six times onto floating-point stack.
fldz
fldz
fldz
fldz
fldz
fld  QWORD PTR [esi-128]    ; Push B[0,j] onto stack.

fld  QWORD PTR [edi-128]          ; Push A[i,0] onto stack.
fmul st(0), st(1)                 ; Multiply A[i,0] by B[0,j].
faddp st(7), st(0)                ; Accumulate contribution to dot product of
                                  ;  A's row i and B's column j.

fld   QWORD PTR [edi+eax-128]     ; Push A[i+1,0] onto stack.
fmul  st(0), st(1)                ; Multiply A[i+1,0] by B[0,j].
faddp st(6), st(0)                ; Accumulate contribution to dot product of
                                  ;  A's row i+1 and B's column j.

fld   QWORD PTR [edi+eax*2-128]   ; Push A[i+2,0] onto stack.
fmul  st(0), st(1)                ; Multiply A[i+2,0] by B[0,j].
faddp st(5), st(0)                ; Accumulate contribution to dot product of
                                  ;  A's row i+2 and B's column j.

fld   QWORD PTR [edi+ebx-128]     ; Push A[i+3,0] onto stack.
fmul  st(0), st(1)                ; Multiply A[i+3,0] by B[0,j].
faddp st(4), st(0)                ; Accumulate contribution to dot product of
                                  ;  A's row i+3 and B's column j.

fld   QWORD PTR [edi+eax*4-128]   ; Push A[i+4,0] onto stack.
fmul  st(0), st(1)                ; Multiply A[i+4,0] by B[0,j].
faddp st(3), st(0)                ; Accumulate contribution to dot product of
                                  ;  A's row i+4 and B's column j.

fmul  QWORD PTR [edi+ecx-128]     ; Multiply A[i+5,0] by B[0,j].
faddp st(1), st(0)                ; Accumulate contribution to dot product of
                                  ;  A's row i+5 and B's column j.
```

The processor can execute the instructions in this code sequence out of order because the instructions are independent. Even though the loads and multiplies are performed sequentially, the floating-point scheduler can execute the FLD and FMUL instructions out of order in addition to the FADD instruction so as to keep the multiplier and adder pipes of the floating-point unit busy. B[0] is initially loaded into an x87 register and multiplied by the loaded elements of each row with the `reg, reg` form of FMUL to minimize the number of load operations that need to be performed. Additionally, the first element from the sixth row of **A** is not loaded but simply multiplied from memory by the loaded element of B[0]. This eliminates an FLD instruction and decreases the number of instructions in the instruction cache and the workload on the processor's decoder. To achieve two floating-point

operations per clock cycle, the number of floating-point operations should be twice the number of load-store operations. In the example above, there are 12 floating-point operations and seven operations requiring loads from memory, so nearly two floating-point operations can be performed per clock cycle.

Align and Pack DirectPath x87 Instructions

The last optimization to be performed is code packing and alignment. Having an abundance of operations in the decoder keeps the processor's schedulers well fed in circumstances where instructions cannot be immediately provided to the decoders. Floating-point x87 code can be aligned to 8-byte boundaries as illustrated here, which is optimal on AMD Family 10h and 12h processors:

```
;Instruction Address        Opcode      Instruction
;================================================
 00000360                   66          DB      066h
 00000361                   DD 06       fld     QWORD PTR [esi]
 00000363                   66          DB      066h
 00000364                   DD 07       fld     QWORD PTR [edi]
 00000366                   D8 C9       fmul    st(0), st(1)

 00000368                   DE C7       faddp   st(7), st(0)
 0000036A                   DD 04 38    fld     QWORD PTR [edi+eax]
 0000036D                   66          DB      066h
 0000036E                   D8 C9       fmul    st(0), st(1)

 00000370                   DE C6       faddp   st(6), st(0)
 00000372                   DD 04 47    fld     QWORD PTR [edi+eax*2]
 00000375                   66          DB      066h
 00000376                   D8 C9       fmul    st(0), st(1)

 00000378                   DE C5       faddp   st(5), st(0)
 0000037A                   DD 04 3B    fld     QWORD PTR [edi+ebx]
 0000037D                   66          DB      066h
 0000037E                   D8 C9       fmul    st(0), st(1)

 00000380                   DE C4       faddp   st(4), st(0)
 00000382                   DD 04 87    fld     QWORD PTR [edi+eax*4]
 00000385                   66          DB      066h
 00000386                   D8 C9       fmul    st(0), st(1)

 00000388                   DE C3       faddp   st(3), st(0)
 0000038A                   DC 0C 39    fmul    QWORD PTR [edi+ecx]
 0000038D                   66          DB      066h
 0000038E                   DE C1       faddp   st(1), st(0)
```

The instruction address specifies the address (in hexadecimal) of the instruction to the right.

Typically three DirectPath instructions occupy 7 bytes. Maintaining 8-byte alignment for the next group of three instructions requires the addition of a single byte. A 1-byte padding can easily be achieved using the single-byte NOP instruction (opcode 90h), as recommended in "Code Padding with Operand-Size Override and Multibyte NOP" on page 68. However, for the special case of x87

instructions, the operand-size override (66h) serves as a high-performance NOP instruction and is the recommended choice for padding an x87 instruction without altering its behavior, as shown here:

```
DB 066h   ; Operand-size override used as high-performance NOP instruction
```

This usage of the operand-size override alone as a filler byte (without an accompanying NOP instruction) is permitted only for x87 instructions. This usage of the operand-size override can be applied to all but four of the x87 instructions. The FLDENV, FRSTOR, FSTENV, and FSAVE instructions and their no-wait forms behave differently when associated with an operand-size override; therefore, these should not be padded with the operand-size override.

10.3 Floating-Point Compare Instructions

Optimization

For branches that are dependent on floating-point comparisons, use the FCOMI, FCOMIP, FUCOMI, and FUCOMIP instructions:

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

The FCOMI, FCOMIP, FUCOMI, and FUCOMIP instructions are much faster than the classical approach using FSTSW. When FSTSW cannot be avoided (for example, backward compatibility of code with older processors), no floating-point instruction should occur between an FCOM, FCOMP, FCOMPP, FICOM, FICOMP, FUCOM, FUCOMP, FUCOMPP, or FTST instruction and a dependent FSTSW instruction. This optimization allows the use of a fast-forwarding mechanism for the floating-point condition codes internal to the processor's floating-point unit and increases performance.

10.4 Using the FXCH Instruction Rather Than FST/FLD Pairs

Optimization

Increase parallelism by breaking up dependency chains or by evaluating multiple dependency chains simultaneously by explicitly switching execution between them.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Although the AMD Family 10h and 12h processor's floating-point unit has a deep scheduler, which in most cases can extract sufficient parallelism from existing code, long dependency chains can stall the scheduler while issue slots are still available. The maximum dependency chain length that the scheduler can absorb is about six four-cycle instructions.

To switch execution between dependency chains, use of the FXCH instruction is recommended because it has an apparent latency of zero cycles and generates only one micro-op. The floating-point unit of the AMD Family 10h and 12h processors contains special hardware to handle up to three FXCH instructions per cycle. Using FXCH is preferred over the use of FST/FLD pairs, even if the FST/FLD pair works on a register. An FST/FLD pair adds two cycles of latency and consists of two macro-ops.

10.5 Floating-Point Subexpression Elimination

Optimization

Reduce the number of superfluous FXCH instructions by putting the shared source operand at the top of the stack to eliminate subexpressions.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

There are cases that do not require an FXCH instruction after every instruction to allow access to two new stack entries. In the cases where two instructions share a source operand, an FXCH is not required between the two instructions. When there is an opportunity for subexpression elimination, reduce the number of superfluous FXCH instructions by putting the shared source operand at the top of the stack—for example:

Examples

Avoid

```
;=====================
; func((x*y),(x+z))
;=====================
fld   x           ; x
fld   y           ; y x
fld   x           ; x y x
fld   z           ; z x y x
faddp st(1), st   ; x+z y x
fxch  st(2)       ; x y x+z
fmulp st(1), st   ; x*y x+z
```

Preferred

```
fld   z           ; z
fld   y           ; y z
fld   x           ; x y z
fmul st(1), st   ; x x*y z
faddp st(2), st   ; x*y x+z
```

10.6 Accumulating Precision-Sensitive Quantities in x87 Registers

Optimization

Accumulate results in the x87 registers rather than the SSE and SSE2 XMM registers, if more than 64 bits of accuracy are required.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

More than 64 bits of accuracy may be required, as when accumulating a result (for example, during the calculation of dot product). The precision of floating-point operations in the x87 registers ST(0)–ST(7) is 80 bits internally, whereas the precision of operations using SIMD instructions is only 64 bits.

Note: *Some compilers may not fully support 80-bit precision.*

10.7 Avoiding Extended-Precision Data

Optimization

Store floating-point data in single-precision or double-precision format.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Loading and storing extended-precision data is significantly slower than storing single- or double-precision data.

Chapter 11 Multiprocessor Considerations

This chapter covers the following topics:

Topic	Page
ccNUMA Optimizations	183
Writing Instruction Bytes to Memory on Multiprocessor Systems	193
Multithreading	194
Data Organization	195
Data Caching	196
False Data Sharing	197
Data-Parallel Threading	198
Stream Processing	199
Multithreaded Libraries	200
Locked Instructions as Memory Barriers	202
Store/Store Barriers in WB Memory	203
Optimizing Inter-Core Data Transfer	203

11.1 ccNUMA Optimizations

AMD Family 10h and 12h multiprocessor systems use cache coherent non-uniform memory access (ccNUMA). For details on optimizing applications for ccNUMA systems, see *Performance Guidelines for AMD Athlon™ 64 and AMD Opteron™ ccNUMA Multiprocessor Systems*, order# 40555.

11.1.1 Scheduling Single and Multithreaded Applications on Multiprocessor Systems

Optimization

On AMD Family 10h and 12h quad-core multiprocessor systems, schedule threads in such a way as to maintain a balanced system load. In most cases, it is advisable to rely on the ccNUMA-aware operating system to make the correct scheduling decisions for single and multi-threaded applications.

Be sure the operating system is properly configured to support ccNUMA. All versions of Microsoft® Windows® XP for AMD64, Windows Server for AMD64 and Windows Vista support ccNUMA. The 32-bit versions of Windows Server 2003 Enterprise Edition and Windows Server 2003 Datacenter Edition require the /PAE boot parameter to support ccNUMA. For 64-bit Linux™, there may be separate kernels supporting ccNUMA that should be selected. The 2.6.x Linux kernels feature

NUMA awareness in the scheduler. Most SuSE and Red Hat 64-bit Linux distributions have the ccNUMA-aware kernel. All versions of Solaris™ for AMD64 support ccNUMA without change.

Application

This optimization applies to:

• 32-bit software

• 64-bit software

Rationale

Some multiple processor systems available today employ a symmetric multiprocessing (SMP) architecture. Processors on an SMP platform generally share a common or centralized memory bus having identical memory access latencies regardless of the processor position. Because the processors use the same bus and memory, system performance may be negatively affected when bottlenecks occur due to increased demands on the single memory bus. Figure 3 shows a simplified diagram of a two processor(2P) SMP system.

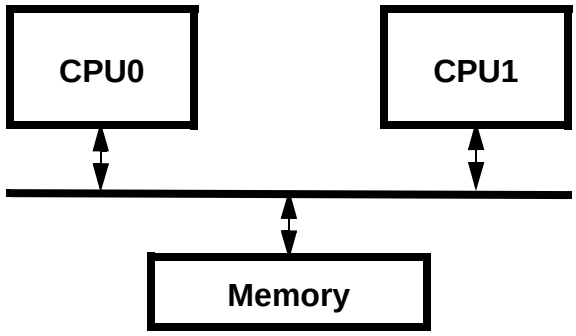


Figure 3. Simple SMP Block Diagram

AMD Family 10h and 12h multiprocessor systems implement cache coherent non-uniform memory access (ccNUMA) architecture to connect two or more processors on the same motherboard. In a ccNUMA design, each processor has its own memory system. In AMD Family 10h and 12h multiprocessor systems, each processor has its own memory controller and its own local memory. When a processor accesses its local memory, the latency is relatively low, especially when compared to that of a similar SMP system. If a processor accesses remote memory—that is, memory located on a different processor—then the access latency is higher. The phrase 'non-uniform memory access' refers to this potential difference in latency. Figure 4 on page 197 shows a simplified diagram of a two processor (2P) AMD Family 10h processor system in a ccNUMA configuration.

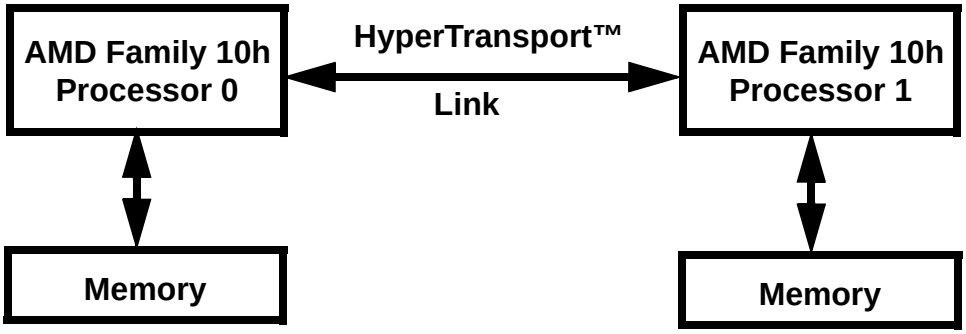


Figure 4. AMD Family 10h 2P System

AMD Family 10h and 12h dual processor systems could have up to four cores on each processor chip that share the on-chip integrated memory controller and memory. Figure 5 shows a simplified diagram of a two processor (2P) quad-core AMD Family 10h system in a ccNUMA configuration.

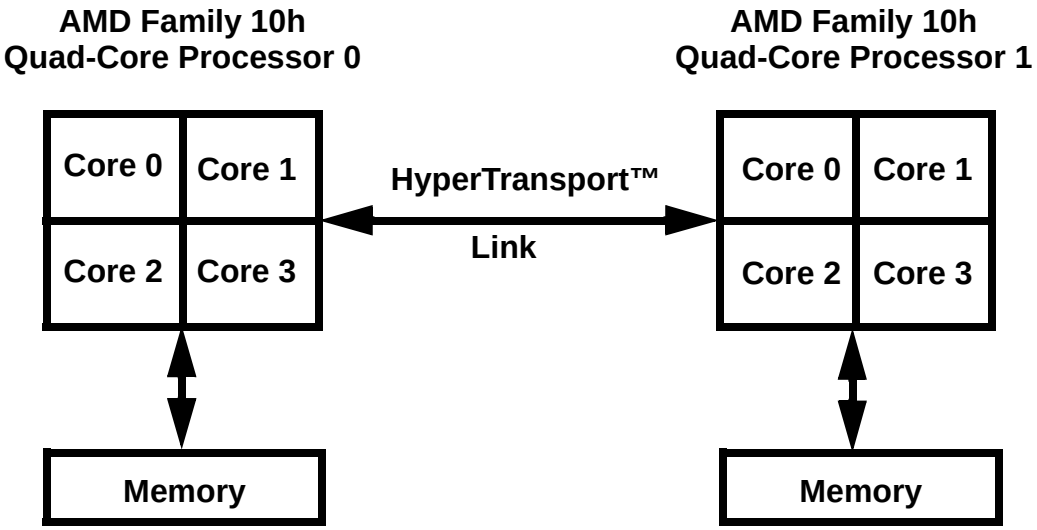


Figure 5. Dual Quad-Core AMD Family 10h Processor Configuration

An operating system running on an AMD Family 10h or 12h platform transparently coordinates and manages memory configuration. Thus, it is not necessary for applications to be aware of memory configuration details. Thanks to the OS, the platform simply appears to have one contiguous block of memory, regardless of how many processors are in the platform. The architecture simultaneously ensures that the entire shared memory space gives consistent values despite potentially parallel accesses from different processors. The phrase “cache coherence” in a ccNUMA system refers to this guaranteed memory consistency.

In an AMD family 10h or 12h (either dual core or quad core) 2P multiprocessor system, each processor is directly connected to the other processor. In addition to the 2P configuration, AMD offers 4P and higher configurations.

Figure 6 shows an example of a four processor quad-core AMD Family 10h system in a ccNUMA configuration. The processors, also called nodes, are numbered N0, N1, N3 and N2 clockwise from the top left. Each node has four cores that are labeled C0, C1, C2 and C3, respectively.

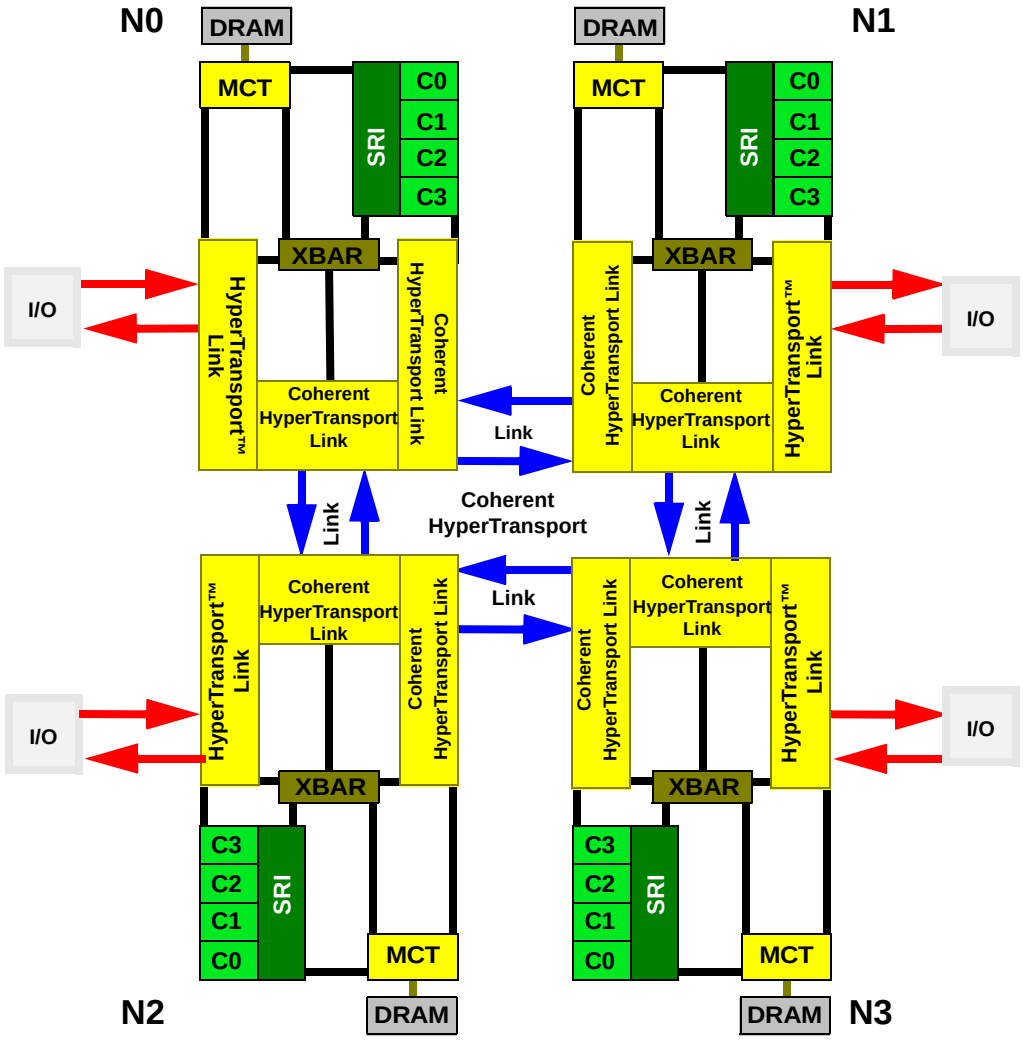


Figure 6. Block Diagram of a ccNUMA AMD Family 10h Quad-Core Multiprocessor System

The four quad-core processors are connected by coherent HyperTransport™ links. Each processor has one bidirectional non-coherent link that is dedicated to I/O and two bidirectional coherent HyperTransport links that each connect to one of the two adjacent quad-core processors in the configuration. In a 4-way configuration, this assures a direct connection for any given quad-core

processor to all the other quad-core processors in the system but one. The throughput of each bidirectional HyperTransport link is 4GB/s in each direction. (This can be platform dependent.) Each node is connected to its own memory.

The term *hop* is commonly used to describe access distances on NUMA systems. If a thread accesses memory on the same node as that on which the thread is running, the memory access is considered a zero-hop access or *local* access. If a thread is running on one node but accessing memory that is resident on a different node, the access is considered a *remote* access. If the node on which the thread is running and the node on which the memory is resident are directly connected to each other, it is a one-hop access. If they are indirectly connected to each other (no direct coherent HyperTransport link) in the four processor configuration shown above, it is considered a two-hop access. For example, if a thread running on Node 0 (N0) accesses memory resident on Node 3 (N3), it is considered a two-hop access.

Figure 7 on page 187 shows the resources inside a node on a multiprocessor system.

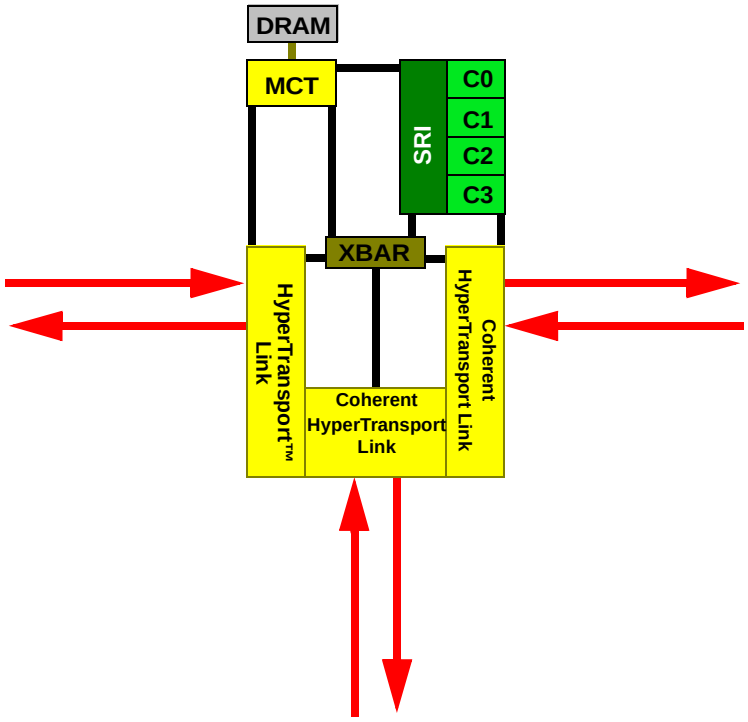


Figure 7. Internal Resources Associated with a Multiprocessor Node

Each node in this system has four cores. The four cores comminicate through a system request interface (SRQ), which in turn talks to a non-blocking crossbar (XBar). The crossbar is connected to the memory controller (MCT) and to the various HyperTransport links. The MCT is connected to the physical memory (DRAM) for that node. The MCT, the SRQ and the XBar on each node all have internal buffers that are used to queue the coherent HyperTransport transaction packets to be transmitted. The SRQ, XBar and MCT collectively make up the Northbridge on the node.

The four cores on each node of the AMD family 10h or 12h processor share the Northbridge, memory and HyperTransport technology resources available on that node. Scheduling should be carried out in such a way as to avoid overloading the resources on a single node, while leaving the resources on the rest of the system unused—in other words, loads should be balanced.

Scheduling multiple threads across nodes and cores of a system is complicated by a number of factors:

- Whether multiple threads access independent data.
- Whether multiple threads access shared data.
- Whether the system is idle.

Multiple Threads-Independent Data

When scheduling multiple threads that access independent data on an idle system, it is preferable, first, to schedule the threads to an idle core of each node until all nodes are exhausted and, then, to schedule the other idle core of each node. In other words, schedule using node major order first, followed by core major order. This is the suggested policy for a ccNUMA aware operating system on an AMD dual-core multiprocessor system.

For example, when scheduling threads that access independent data on a four-way quad-core AMD family 10h or 12h system, scheduling the threads in the following order is recommended (see Figure 6 on page 186):

- Core 0 on node 0, node 1, node 2 and node 3 in any order
- Core 1 on node 0, node 1, node 2 and node 3 in any order
- Core 2 on node 0, node 1, node 2 and node 3 in any order
- Core 3 on node 0, node 1, node 2 and node 3 in any order

Multiple Threads-Shared Data

When scheduling multiple threads that share data on an idle system, it is preferable to schedule the threads on both cores of an idle node first, then on both cores of the the next idle node, and so on. In other words, schedule using core major order first followed by node major order.

For example, when scheduling threads that share data on a four-way quad-core AMD family 10h or 12h system, AMD recommends using the following order:

- Core 0, 1, 2, or 3 on node 0 in any order
- Core 0, 1, 2, or 3 on node 1 in any order
- Core 0, 1, 2, or 3 on node 2 in any order
- Core 0, 1, 2, or 3 on node 3 in any order

Scheduling on a Non-Idle System

It is a more difficult task to schedule multiple threads optimally for an application on a non-idle system. It requires that the application make global holistic decisions about machine resources, coordinate itself with other applications already running, and balance decisions between them. In such cases, it is better to rely on the OS to do the appropriate load balancing. In general, most developers will achieve good performance by relying on the ccNUMA-aware OS to make the right scheduling decisions on idle and non-idle systems.

In addition to the scheduler, several NUMA-aware operating systems provide tools and APIs to allow the developer to explicitly bind a thread (set thread affinity) to a certain core or node. Using these tools or APIs overrides the scheduler and hands over control for thread placement to the program. For additional details on the thread/process affinity tools and APIs supported in various OSs, refer to Appendix D, “Tools and APIs for AMD Family 10h and 12h ccNUMA Multiprocessor Systems,” on page 289.

11.1.2 Data Locality Considerations on Multiprocessor Systems

Optimization

Keep data accessed by a thread local to the node on which the thread runs. In a multithreaded application in which each thread operates on largely independent data, each thread should allocate and initialize the data it accesses and allow the ccNUMA-aware operating system to make the right data locality decisions.

In multithreaded applications, performance may benefit from taking advantage of API functions or tools for thread and memory placement (thread and memory affinity) offered by the operating system.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

It is best to keep data local to the node from which it is being accessed. Accessing data remotely is slower than accessing data locally. The further the hop distance to the data, the greater the cost of accessing remote memory. For most memory latency-sensitive applications, keeping data local is the single most important recommendation to consider.

Almost all ccNUMA-aware operating systems by default rely on the first-touch policy: the physical memory for data is only committed on the node on which the thread or process writing to it for the first time runs. Commitment implies mapping of virtual pages to zeroed out physical pages. This is done by the OS when it detects a first-touch and takes a page fault. (Windows Vista is the exception.

For additional details, refer to “NUMA Optimization in Windows Applications” at Develop with AMD (http://developer.amd.com/articlex.jsp?id=106.) Thus, data is kept local on the node where the thread or process that writes to it for the first time is run.

The OS keeps data local on the node where first-touch occurs as long as there is enough physical memory available on that node. If enough physical memory is not available on the node, then different OSs use various advanced techniques to determine where to bind the data.

Memory once bound to a node by the first touch policy normally resides on that node for its lifetime. However, the OS scheduler could migrate the thread or process that first touched the memory from one core to another core even on a different node. This can be done by the OS for the purpose of load balancing.

This can move the process/thread farther from its allocated memory. Most schedulers will try to bring the thread or the process back to the core on which the thread was previously running and on which its memory was local, but this is not guaranteed. Furthermore, the thread or process can dynamically allocate and first-touch more memory on the node to which it was moved before it is moved back. This is a difficult problem for the OS to resolve, since it has no prior information as to how long the thread or process is going to run and, hence, whether migrating it back is optimal or not.

If an application demonstrates that threads are being moved away from their associated memory by the scheduler, it is typically useful to explicitly set thread placement. By explicitly pinning a thread to a node, the application can tell the OS to keep the thread on that node and, thus, keep data accessed by the thread local to it by the virtue of the first touch policy.

The performance improvement obtained by explicit thread placement may vary depending on whether the application is multithreaded, whether it needs more memory than available on a node, whether threads are being moved away from their data, etc.

In cases in which threads are scheduled from the outset on a core that is remote from their data, it might be useful to explicitly control data placement. This is discussed in detail in the “Scheduling on a Non-Idle System” on page 189. Advanced software developers can refer to Appendix D, “Tools and APIs for AMD Family 10h and 12h ccNUMA Multiprocessor Systems,” on page 289 for additional details on support for these tools and APIs in various OSs.

11.1.3 Techniques to Minimize and Alleviate Data Sharing

Optimization

Avoid accessing data in memory that was first touched by a thread running on a different node.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

When data is shared between threads running on different nodes, the default policy of local allocation by first touch used by the OS can become non-optimal.

For example, a multithreaded application may have a startup thread that sets up the environment, allocates and initializes a data structure and forks off worker threads. As per the default local allocation policy, the data structure is placed in the physical memory of the node where the start up thread performed the first touch. Forked worker threads are then spread around by the scheduler to be balanced across all nodes and their cores. A worker thread starts accessing the data structure remotely from the memory on the node where the first touch occurred. This scenario could lead to significant memory and HyperTransport traffic in the system, with the node where the data resides becoming the bottleneck. This situation is especially bad for performance, firstly, if the startup thread only performs the initialization and afterwords no longer needs the data structure and, secondly, if only one of the worker threads needs the data structure. In other words, the data structure is not truly shared between the worker threads.

It is best in this case to use a data initialization scheme that avoids incorrect data placement due to first touch. This is done by allowing each worker thread to first-touch its own data or by explicitly pinning the data associated with each worker thread on the node where the worker thread runs.

Certain OSs provide memory placement tools and APIs that also permit data migration. A worker thread can use these to migrate the data from the node where the start up thread performed the first touch to the node where the worker thread needs it. There is a cost associated with the migration and it would be less efficient than using the correct data initialization scheme in the first place.

If it is not possible to modify the application to use a correct data initialization scheme or if data is truly being shared by the various worker threads—as in a database application—then a technique called node interleaving can be used to improve performance. Node interleaving allows for memory to be interleaved across any subset of nodes in the multiprocessor system. When the node interleaving policy is used, it overrides the default local allocation policy used by the OS on first touch.

Let us assume that the data structure shared between the worker threads in this case is of size 16 KB. If the default policy of local allocation is used, then the entire 16 KB data structure resides on the node where the startup thread does first touch. However, using the policy of node interleaving, the 16-KB data structure can be interleaved on first touch such that the first 4 KB ends up on node 0, the next 4 KB ends up on node 1, and the next 4 KB ends up on node 2 and so on. This assumes that there is enough physical memory available on each node. Thus, instead of having all memory resident on a single node and making that the bottleneck, memory is now spread out across all nodes.

The tools and APIs that support explicit thread and memory placement mentioned in the previous sections can also be used by an application to use the node interleaving policy for its memory. (See Appendix D, “Tools and APIs for AMD Family 10h and 12h ccNUMA Multiprocessor Systems,” on page 289.)

By default, the granularity of interleaving offered by the tools/APIs is usually set to the size of the virtual page supported by the hardware, which is 4 K (when system is configured for normal pages,

which is the default) and 2 M (when system is configured for large pages). Therefore any benefit from node interleaving will only be obtained if the data being accessed is significantly larger than a virtual page size.

If data is being accessed by three or more cores, then it is better to interleave data across the nodes that access the data than to leave it resident on a single node. We anticipate that using this rule of thumb could give a significant performance improvement. However, developers are advised to experiment with their applications to measure any performance change.

11.1.4 Keep Locks Cacheable and Aligned to a Cache Line Boundary

Optimization

In general, it is good practice for user-level and kernel-level code to keep locks aligned to their natural boundaries. In some hardware implementations, locks that are not naturally aligned are handled with the mechanisms used for legacy memory mapped I/O and should absolutely be avoided if possible.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

If a lock is aligned properly, it is treated as a faster cache lock. The significantly slower alternative to a cache lock is a bus lock, which should be avoided at all costs. Bus locks are very slow and force serialization of many operations unrelated to the lock within the processor. Furthermore, bus locks prevent the entire HyperTransport fabric from making forward progress until the bus lock completes. Cache locks on the other hand are guaranteed atomicity by using the underlying cache coherence of the ccNUMA system and are much faster.

11.1.5 Cache-Line Data Sharing

In a ccNUMA multiprocessor system, data within a single cache line that is shared between cores, even on the same node, can reduce performance. In certain cases (for example, semaphores), this kind of cache-line data sharing cannot be avoided, but it should be minimized where possible.

Application

This optimization applies to:

- 32-bit software

- 64-bit software

Rationale

Data can often be restructured, so that this does not occur. Cache lines on AMD Family 10h and 12h processors are presently 64 bytes, but a scheme that avoids this problem regardless of cache-line size makes for more performance-portable code.

For example, per-thread data can be allocated on the heap (for example, by means of calls to malloc()); this is preferred over statically defined shared arrays and variables that are potentially located in a single cache line. Furthermore, some software environments even provide special versions of malloc() that guarantee data alignment to a specified value (size). These can be useful in aligning data and eliminating unwanted cache line overlap.

11.2 Writing Instruction Bytes to Memory on Multiprocessor Systems

A common situation in dynamically optimized applications is that in which a thread on one processor in a multiprocessor system (which we will call the writer) is required to replace an original code segment with some new code segment, while there are one or more other threads (executors) on other processors that could possibly execute the original code. This can occur, for example, when a function is recompiled or reoptimized at run time.

For simplicity, this section discusses the case in which the original code consists of a single instruction. If the original code consists of multiple instructions, the writer must always ensure in some way that an executor is not in the middle of the original code.

Rule 1

If the part of the original code that needs to be patched fits within an aligned 8-byte boundary, then no special considerations are necessary. The writer may simply store the new code into memory.

In the following example, the instruction itself crosses an aligned 8-byte boundary but since the first byte is not changing, the part to be changed does not cross an aligned 8-byte boundary and so can be changed with a single store.

```
Original Code xxxxxF: E8 78 56 34 12 Call $+12345678
New Code xxxxxF: E8 44 33 22 11 Call $+11223344
```

When a modification does cross an aligned 8-byte boundary, then care must be taken that the executor not see an invalid combination of the original code and the new code. There is no architectural store instruction, including instructions that use the lock prefix, to ensure that an executor will not see a combination of the original code and the new code. Instead, one of the following methods can be used:

- Software semaphores can be used between the writer and the executors to prevent executors from entering the original code

or

- The original code can be modified in stages by first writing a branch at the beginning of the original code to catch an executor and then modifying the remaining code. In this case a systemdependent delay must be used after writing the branch. This delay is necessary to ensure that any executor that had already fetched the first bytes of the original code (before the branch was written) has finished fetching the rest of the original code.

To modify in stages, the writer uses the following steps:

1. Modify the beginning of the original code with a branch that will catch any executor that enters the original code. The easiest branch to use is the two-byte short JMP to self (bytes EB, FE). This requires that the first two bytes of the original code not cross an 8-byte boundary. When the original code is generated and the compiler knows that it is a candidate for patching, it is recommended that a NOP be inserted. If the first two bytes of the original code do cross an 8-byte boundary, a one-byte BPT instruction and a special BPT trap handler that returns to the BPT instruction must be used.
2. Wait for a system-dependent delay. This delay ensures that any Executor who had fetched the beginning of the Original Code before the branch was written has finished fetching the rest of the Original Code.

- The maximum amount of delay can be lessened by avoiding patches across a 4K byte page boundary and lessened further by avoiding patches which cross a 64 byte cache-line boundary.

3. Leaving the self-branch in place, modify the rest of the original code with the new code.

4. Replace the self-branch with the corresponding bytes of the new code.

11.3 Multithreading

The subject of creating multithreaded software is quite broad, and many resources exist that address its various aspects. Here we briefly discuss those aspects of multithreading that are most relevant from a hardware perspective.

To fully utilize the CPU power of multicore processors, applications must implement *scalable* threading. In other words, the application must be able to partition the work load into a variable number of threads, to match the available resources on the particular machine.

Many of problems can be resolved by the implementation of various programming practices, including *task decomposition*, *careful data organization*, and *data caching and sharing*. Two of the most important and straightforward ways to implement scalable threading are by means of *data-parallel threading* and *stream processing*. These methods are described in detail in the followuing sections.

11.3.1 Task Decomposition

Optimization

For each task, use multiple threads in parallel to process equal workloads involving different data items.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Historically, multithreading has been used to implement separate functions. For example, one thread might perform I/O operations while another thread handles user input. This approach, called functional threading or task-parallel threading, can sometimes simplify the structure of a program, especially when the program is performing several asynchronous tasks.

However, functional threading has limitations. Only a fixed, limited number of threads are used. Also, the workloads in different threads are not balanced. For these reasons, functional threading is not a good match for present and future multicore processors. It doesn't scale up to utilize the hardware.

A much better approach is data-parallel threading. In data-parallel threading, each CPU-intensive task is handled in sequence. For each task, multiple threads are used in parallel to process equal workloads involving different data items. Ideally, the application can use *N* threads, to match an *N*-core processor or system. Not every processing task can be implemented using data-parallel threading. For example, data decompression and decryption are often inherently sequential tasks.

11.3.2 Data Organization

Optimization

Divide data cleanly into many largely independent sets.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Clearly, data parallel threading requires a certain class of algorithms. For example, if data is organized as a single linked list, the operation of accessing the list is not well suited to multithreading. On the other hand, an array of uniformly sized structures can usually be accessed in parallel as *N* chunks.

Double buffering can be used to good effect. Creating one set of data which is 100% "read only" can be valuable, even if this involves total replication of data sets in memory. Processing can read one copy of the data, while writing to the other copy. This can greatly reduce or eliminate cache thrashing and race conditions. Copying all the data might not be a performance win if you are just running two threads, but it can pay off as the number of threads grows.

11.3.3 Data Caching

Optimization

Make good use of the data caches.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Single-threaded applications are influenced by cache effects. The processor stores recently used data in a local cache memory, making subsequent operations on that data run faster. All of the traditional cache factors apply to multithreaded code: limited cache size, data replacement policy, how many cache ways the cache implements, L1 vs. L2 cache, and related criteria. (For additional information on cache architecture and optmizations, see section 7.5, “Memory Caches” in the *AMD64 Architecture Programmer’s Manual: Volume 2 System Instructions* (order# 24593), and Chapter 5, “Cache and Memory Optimizations” on page 71. Two additional factors enter the picture when multiple threads are running on multiple cores: data sharing between caches, and false sharing between caches.

11.3.4 Data Sharing between Caches

Optimization

Design threads so that each thread operates on separate data.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

All threads in a process have a coherent view of memory. If certain data is used by multiple threads, then every time that data is modified, it must be copied into more than one cache. This data copying is avoided if threads are designed so that each thread operates on separate data. Of course, if threads are only reading the data and not modifying it, they can all safely share the same data.

11.3.5 False Data Sharing

Optimization

To avoid false data sharing, keep each thread's data carefully separate by enforcing, for example, 64-byte alignment during allocation.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

False sharing is a subtle variation on data sharing. The data cache is managed on a cache-line basis, where each naturally aligned 64-byte cache line is treated as a unit. If any byte is modified, the entire cache line is tagged as modified. So if multiple threads access different parts of the same cache line, and at least one thread is modifying the data, that cache line must be copied into the other caches to maintain coherence. The threads are functionally independent, but they incur a performance penalty as if they were actually sharing data. False sharing can be avoided by keeping each thread's data carefully separate, for example by enforcing 64-byte alignment during allocation.

Clean data separation at the algorithm level will minimize the occurrence of real or false data sharing.

11.3.6 Data-Parallel Threading

Optimization

Break up a workload into multiple data sets and use threads to perform the same operations on different data in parallel.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

Data-parallel threading involves breaking up a workload into multiple data sets and using threads to perform the same operations on different data in parallel. The data organization and algorithms used by the application must be carefully designed to efficiently support this form of parallelism to avoid race conditions or expensive synchronization mechanisms.

Data-parallel threading can usually be made to achieve very good load balancing between threads, efficiently utilizing all available CPU resources. Furthermore, if an application is designed for data-parallel threading, the threads do not alter the overall logical order of operations in the application, so they do not introduce as many potential deadlocks or race conditions that can complicate other threading strategies.

A trivial example of data-parallel threading involves the addition of two arrays of numbers. The basic operation might be expressed in C++ as:

```
for (int i=0; i < 10000; i++) {
  c[i] = a[i] + b[i];
}
```

A data-parallel multithreaded implementation would process the arrays in chunks, for example if four threads are used, the values of array index *i* would be assigned to each of the four threads as follows:

thread #0	i = 0 through 2499
thread #1	i = 2500 through 4999
thread #2	i = 5000 through 7499
thread #3	i = 7500 through 9999

Threading can be implemented explicitly, for example, using Win32 thread APIs on Windows® or Pthreads on Linux®. The application must choose how many threads to run, create and/or start the threads, and detect their completion. The application is also typically responsible for detecting the number of processors available at run time.

Most modern compilers also support OpenMP, which greatly simplifies the syntax for data-parallel threading in loops. For the following loop, using OpenMP requires just one extra line of code (a pragma), which partitions the workload across an appropriate number of threads:

```
#pragma omp parallel for
for (int i=0; i < 10000; i++) {
  c[i] = a[i] + b[i];
}
```

OpenMP has a simple API, and it supports many options for controlling how the data-parallel threading is executed. For details see http://www.openmp.org.

On platforms that support multiple CPU nodes (as opposed to simply supporting one multicore CPU), additional performance gains can be achieved because of greater system memory bandwidth available. Memory buffers for storing thread-specific data should be allocated locally to the NUMA node, by calling the allocation function from within the thread. In some heavily threaded applications, it also makes sense to set thread affinity at the time of thread creation to distribute them across multiple NUMA nodes. Care must be taken when manually setting affinity. Read more about NUMA optimization in “ccNUMA Optimizations” on page 183, above, and in http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/40555.pdf and other papers on the Develop with AMD Website.

11.3.7 Stream Processing

Use stream processing to operate on large arrays of related data.

Application

This optimization applies to:

- 32-bit software
- 64-bit software

Rationale

With the advent of truly programmable graphics processing units (GPUs), the programming paradigm of stream processing has become much more relevant. Strictly speaking, stream processing is not a form of multithreading, but it shares many of the same constraints on data organization and algorithm choice as data-parallel threading does.

In stream processing, a set of kernels (i.e., functions) operate on *streams*—large arrays of related data. Typically, kernels implement math operations that can be vectorized, for instance, by using vector SSE instructions for best performance. For maximum efficiency, kernels consume streams that are generated as output by other kernels; these streams persist locally in the low-latency CPU data cache, instead of making a trip through system memory.

At the appropriate time, streams are explicitly moved between CPU cache and system memory, as a logically separate process from the kernel operations, so data movement is only loosely coupled to processing. In principle, this decoupling can enable more efficient gather/scatter operations on blocks of data that comprise the streams. For maximum efficiency, stream data should be organized contiguously in memory. In many cases, for best performance the stream data can be read from memory using software prefetch instructions and, finally, written back to memory using the streaming store instructions, which avoid disturbing the L2 cache.

If the application's algorithms and data structures are mappable onto the stream/kernel model, then a stream processing approach can be profitably implemented. This can result in increased CPU performance because data cache locality and memory bandwidth are well utilized and also because data-parallel threading can usually be employed in conjunction with the stream processing. Perhaps even more importantly, organizing an application to fit the stream processing model can pave the way for off-loading the heavy computational workloads to a highly parallel GPU chip or other specialized processor.

11.3.8 Multithreaded Libraries

Programmers can use multithreaded code libraries, such as the AMD Performance Library (APL) and AMD Core Math Library (ACML), to great advantage in writing applications that incorporate the multithreading paradigm.

11.4 Memory Barrier Operations

Memory barriers of type A/B, where A and B represent either a load or store memory operation and A is ordered prior to B in program order, allow the programmer to specify that *older* memory operations of type A (load or store) cannot appear to be passed by any *younger* memory operations of type B (load or store). Here, B *passing* A means that although A precedes B in program order, the results of instructions A and B may be returned in any order.

There are four types of memory barriers:

- Load/Load—older loads are not passed by younger loads.
- Store/Store—older stores are not passed by younger stores.
- Load/Store—older loads are not passed by younger stores.
- Store/Load—older stores are not passed by younger loads.

Memory Barriers in WB Memory

On the AMD64 architecture, when using writeback (WB) type memory without streaming stores, the only type of barrier that requires an explicit barrier instruction is Store/Load. When streaming stores are used, the Store/Store barrier also requires an explicit barrier instruction. In WB memory, all other barriers are implicit in the AMD64 architecture. For additional information on memory and memory barrier instructions, see “Forcing Memory Order” in the *AMD64 Architecture Programmer’s Manual*

Volume 1: Application Programming and Chapter 7, “Memory System” in the *AMD64 Architecture Programmer’s Manual Volume 2: System Programming*.

Memory barriers in WB memory are unnecessary in systems consisting of a single processor core.

Store/Load Barriers in WB Memory

On the AMD64 architecture there are three ways to achieve the Store/Load barrier in WB memory (see section 3.9 in APM volume 1):

- MFENCE instruction
- A locked instruction that reads and writes memory—any instruction of the form LOCK *op mem, reg* or LOCK *op mem,imm*. (The specific instruction XCHG *mem, reg* is treated as locked whether or not a LOCK prefix is used.)
- Architecturally serializing instructions such as CPUID.

11.4.1 Locked Instructions as Memory Barriers

Optimization

Use locked instructions to implement Store/Load barriers.

Application

Applies to programs running on multicore processors or on multiple single core processors.

Rationale

On AMD Family 10h and 12h processors, the MFENCE instruction is a serializing instruction. This stalls the pipeline and the processor cannot begin processing any further instructions until all *previous* instructions are completed and any outstanding memory operations (such as prefetches and stores) have completed. Architecturally serializing instructions such as CPUID have the same pipeline stall behavior as MFENCE. The LOCKed instructions do not stall the pipeline and, thus, allow more parallelism.

LOCKed instructions that access shared memory (memory shared between processor cores) incur a delay while the cache line is changed to modified state and data is (potentially) transfered between caches in the system. LOCKed instructions that are not naturally aligned incur the very high overhead of a bus lock.

When possible, make the LOCKed instruction perform a useful store (an XCHG *mem,reg* instruction can be used for this purpose, assuming the *reg* can be overwritten). The memory location that is the target of the store (e.g., XCHG) instruction should be in the exclusive state in the processor core's local L1 cache. Avoid using a memory location that is shared with other processor cores (even if it is only written by the local processor core). It is also very important to avoid using a memory location that is not naturally aligned.

Thus, for a pattern such as:

```
mov   localmem2, rax          ;; store to local memory
mov   sharedmem1, rbx         ;; store to shared memory
StoreLoad_Barrier
mov   rcx, sharedmem3         ;; load from shared memory
```

Preferred:

```
mov   sharedmem1, rbx
xchg  localmem2, rax   ;; performs local store and StoreLoad barrier
                       ;; in one instruction (note: modifies rax)
mov   rcx, sharedmem3
```

Avoid:

```
mov   localmem2, rax
```

```
xchg   sharedmem1, rbx   ;; avoid using shared mem for locked operation
mov   rcx, sharedmem3
```

Avoid:

```
mov   sharedmem1, rbx
mov   localmem2, rax
mfence                   ;; avoid MFENCE which is serializing
mov   rcx, sharedmem3
```

When the locked instruction cannot be made to do a useful store, there are several variations of LOCK *op mem*, *imm* that do not modify the memory contents or any registers other than the FLAGS register, for example:

```
LOCK OR DWORD PTR localmem, 0
```

To repeat, the memory location that is the target of the locked instruction should be in the exclusive state in the processor's local L1 cache. A location on the stack such as [RSP] in 64-bit mode or [ESP] in 32-bit mode generally meets this criteria. To avoid Store-To-Load forwarding issues, the location should be addressed using the same data width with which it is otherwise accessed.

11.4.2 Store/Store Barriers in WB Memory

When performing a Store/Store in WB memory, a Store/Store barrier is only required when using streaming stores and then only in systems with more than one processor core. Store/Store barriers can be achieved in one of the following ways (see section 3.9, “Memory Optimization” in *AMD64 Architecture Programmer’s Manual Volume 1: Application Programming*):

- Using the SFENCE instruction
- Using any of the methods covered under the topic of Store/Load barriers.

Optimization

Use the SFENCE instruction to implement a Store/Store barrier.

Application

Applies to programs running on multicore processors or on multiple single core processors.

Rationale

The SFENCE instruction is not a serializing instruction, so it achieves the desired effect of waiting for the write-combining buffers to drain, while allowing parallelism with other non-store instructions.

11.5 Optimizing Inter-Core Data Transfer

AMD Family 10h and 12h processors are the first to incorporate four distinct cores on a single die and the first to have a cache that all the cores share. This shared cache provides an efficient way to

handle a group of computional problems belonging to a producer/consumer model: a program thread running on one core produces data that is intended for consumption by a thread that is running on another core. In such cases, round trips to and from main memory can be avoided by arranging for pairs of cores to communicate through the shared cache.

A naïve implementation of a producer/consumer program on AMD Family 10h and 12h processors will produce bandwidth results that appear to be throttled by main memory speeds. Main memory speeds can vary, but with DDR2 533 memory (average grade), this is around ~4 GB/s. Yet, with some knowledge of AMD Family 10h and 12h cache architecture, it is possible to boost throughput approximate 3x. (These numbers represent the aggregated bandwidth from all four cores transferring data to each other, or two producer/consumer pairs on a four core processor.)

The producer/consumer program is handled by setting up a system in which the producer and consumer threads chase each other around a ring buffer through which they communicate and share data. When a thread reaches the end of the buffer, it wraps back around to the beginning and keeps reading or writing. There are three constraints on such constructs:

- The consumer thread must lag the producer thread by an amount determined by the size of the L1 + L2 caches. The ring buffer cannot be smaller than the minimum distance by which the threads must be separated or the producer thread will not have time to spill its data into the L3 cache when the consumer wants to read it, causing a cache miss for the consumer.
- There is similar constraint on the distance that the producer thread must lag behind the consumer thread. This is explained below.
- If the ring buffer is of significant size, as explained below, the producer thread must not lead the consumer thread by too much or the data it places in the L3 cache will more than likely be evicted to main memory before the producer thread wraps back around again.

These caveats are discussed in the following sections.

Cache sizes and thread distances

The AMD Family 10h and 12h processors have three levels of cache, a 64KB L1, a 512KB L2, and a 2MB or greater L3 (10h only). The lower L1 and L2 caches are private to the individual cores; The highest level L3 cache (10h only) is shared by all four cores. AMD Family 10h cache design is a “mostly exclusive” “victim” cache.

“Mostly” exclusive cache hierarchies. In an *exclusive* cache hierarchy, only one copy of the data exists anywhere in the entire cache hierarchy. In such a system, when a thread hits in the L3 cache, it moves the cache line to a lower level of cache (which is private to the particular core running the thread) without leaving a copy in the L3 cache. However, in AMD Family 10h processors, for purposes of optimizing for multiple readers, multiple cores may generate local copies when they access the shared L3 cache line. For this reason, the cache is termed “mostly” exclusive.

Victims. Bringing a chunk of new data into a cache requires allocation of cache space by displacing an older chunk, the so-called "victim," from that cache. This victim cache line is then pushed into the next higher level cache in the hierarchy; of course this, in turn, triggers the displacement of a victim

from the higher level cache. For this reason, the L3 (shared) cache is only filled with victims from the L2 cache. When a thread running on a specified core fills the local L1 cache, no data is reflected into the L3 cache.

Thread sizes and distances. When the producer thread produces a cache-line of data, it exists only in the L1 cache of the producer core; it does not get written-through to the L2 or L3 caches. When the L1 cache is full, the old data (victim) gets pushed up to the L2 cache; likewise, when the L2 cache is full, data is evicted to the L3 cache. For a program to maximize the bandwidth between two cores through the processor L3 cache, the lag distance between the consumer threads and the producer threads must be greater than the sum of sizes of the L1 and L2 caches (64K + 512K = 576K), so that the consumer core can read victim blocks from the producer core.

However, the producer thread cannot get too far ahead of the consumer thread, or it will flood the L3 cache with victims and start spilling unread cache-lines to memory. On a four core processor, two producer/consumer pairs can run simultaneously, so the most optimal allocation would be to divide up the L3 cache evenly between pairs; for example, on current AMD microprocessors having a 2MB L3 cache, each pair should be allocated 2MB/2 = 1MB of L3 cache. Therefore, the producer thread should be at least 576K ahead of the consumer, but no more than 1024K + 576K = 1600KB or 1.56MB or it will evict its own data from the L3 cache. If only one producer/consumer pair is running on the processor, the entire L3 cache can be dedicated to the pair.

MOESI protocol issues

Even if the abovementioned distance constraints are followed, the measured bandwidth will still be limited by the performance of the DRAM controller. This results from the subtle interaction of the cache lines as they are touched by the producer and consumer threads and the MOESI protocol that AMD Family 10h and 12h processors implement to maintain cache coherency. (For a complete description of the MOESI protocol, see the (*AMD64 Architecture Programmer's Manual Vol 2: System Programming*, order# 24593.)

The MOESI cache coherency protocol is defined by the state of data in a cache line in relation to other copies of the data (in memory, another processory cache, etc.). These states are summarized as follows:

- Modified(M)—The cache line holds the most recent correct copy of the data and the copy in memory is stale. (No other copies exist.)
- Owned (O)—A cache line in the owned state holds the most recent, correct copy of the data, which may be shared by other processors. This copy is responsible for updating main memory, when evicted. (The copy in memory may be stale and other processors may hold a copy in the S state.)
- Exclusive (E)—A cache line holds the most recent, correct copy of the data, which is identical to the copy in memory. (No other processor holds a copy of the data.)
- Shared (S)—A cache line in the shared state holds the most recent, correct copy of the data, which may be shared by other processors. (The copy in memory is stale.)

- Invalid (I)—A cache line does not hold a valid copy of the data. (valid copies are in main memory or another processor cache.)

The producer thread allocates cache-lines in the modified (M) state, as an automatic consequence of writing a new entry. Eventually, these M-marked cache lines will start to fill the L3 cache, thanks to the adherence to the abovementioned rules defining the allowable distance between threads. When the the consumer reads the cache line, the MOESI protocol changes the state of the cache line to owned (O) in the L3 cache and pulls down a shared (S) copy for its own use. Now, the producer thread circles the ring buffer to arrive back to the same cache line it had previously written. However, when the producer attempts to write new data to the owned (marked 'O') cache line, it finds that it cannot, since a cache line marked 'O' by the previous consumer read does not have sufficient permission for a write request (in the MOESI parlance). To maintain coherence, the memory controller must initiate probes in the other caches (to handle any other S copies that may exist)—and this is slow.

Thus, it is preferable to keep the cache line in the 'M' state in the L3 cache. Then, when the producer comes back around the ring buffer, it finds the previously written cache line still marked 'M', to which it is safe to write without coherency concerns.

The PREFETCHW instruction provides the means to control this cache allocation restriction. The PREFETCHW instruction provides a hint to the processor that the program intends to modify the cache line, so the processor keeps the cache line in the 'M' state. To clarify how this works, we will step through a scenario in which the consumer thread uses PREFETCHW to proactively fetch cache lines.

As previously mentioned, the producer thread first spills a cache line marked 'M' into the L3 cache. At some later time, the consumer thread calls the PREFETCHW instruction to load the cache line into the L1 data cache; this time, the processor keeps the cache line marked 'M', removes the cache line from L3 and pulls the line down the cache hierarchy, into the consuming core's L1 cache. No 'S' copy is handed to the consumer core, and no 'O' copy remains in the L3 cache. As far as the producer core is concerned, the cache line is gone.

If the producing thread were to wrap around the buffer and attempt to write to the cache line, it would register as a cache miss and a request would be sent to the memory controller. To avoid this, the consumer needs to evict the cache line back out to the processor shared L3. As discussed previously, the only way to acheive this is to have the cache line trickle back up the cache hierarchy, until it eventually becomes a victim block into the L3 cache again; in other words, the consumer reads enough memory to equal its L1 + L2 cache size (576K), forcing the cache line to evict before the producer thread needs to write to that memory. When that happens, it's as if the producer never knew that the cache line was gone. When it writes new data to the cache line and it finds it in the L3 cache; the producer and consumer are communicating through the L3 cache, fully utilizing the inherent speed and bandwidth.

To assure that the consumer has enough time to spill its contents into the L3 cache, the producer thread must lag the consumer thread by a distance that is at least equal to the sum of the sizes of the L1 + L2 cache in modulo arithmetic. This has further implications for the size of the ring buffer. If the consumer must lag the producer by *X* bytes, and the producer must lag the consumer by *X* bytes, then

the buffer must be at least $2 \times X$ in size to achieve maximum performance. In practice, it is not a bad idea to pad the numbers by at least two units of granularity to allow for some extra space, since cache eviction is not controllable or precise. For instance, assuming that the producer and consumer threads write data in 16K chunks before checking their positions relative to each other, it would be safe to have the producer and consumer enforce a distance of ((L1 + L2) + $2 \times$*granularity*) or (64K + 512K) + $2 \times$16K = 608K apart. This in turn implies that the ring buffer size should be at least 1216K.

Summary

A producer/consumer program can achieve harmony by successfully bouncing an ‘M’ marked cache line back and forth between the consuming and producing threads through the fast L3 cache. A quick review of the constraints to keep in mind to achieve this behavior follows:

- The consumer thread needs to ‘lag’ the producer thread by at least L1 + L2 cache size (modulo arithmetic).
- The producer thread needs to lag the consumer thread by at least L1 + L2 cache size (modulo arithmetic).
- The ring buffer should be at least $2 \times$(L1 + L2).
- The producer thread should not get so far ahead of the consumer to flood the L3, if larger ring buffers are used.
- Use PREFETCHW on the consumer side, even if the consumer will not modify the data.
- Add a small extra factor to the calculated sizes to give the threads additional space when communicating through the caches.

In general, the AMD Family 10h and 12h cache is optimized for widely shared data, i.e. one core produces data that may be of interest to several other cores. The ‘S’ and ‘O’ states provide coherence for multiple readers of the same data. One core is responsible for the data in the ‘O’ state, but that data can be safely shared with many other cores through the ‘S’ state. In the producer/consumer program however, it is known ahead of time that the data the producer creates is only interesting to the matching consumer thread, and not to any other thread. Following the constraints listed above, it is possible to achieve an approximate 3x increase in throughput for two producer/consumer pairs on AMD Family 10h and 12h processors.

Chapter 12 Optimizing Secure Virtual Machines

The goal of this chapter is to enable virtual machine monitor (VMM) software engineers to minimize the performance overhead imposed by the virtualization of a guest. A significant consumer of processor cycles on microprocessors enabled for AMD Virtualization™ (AMD-V™) is the world switch, which refers to the process of running either a VMRUN instruction to enter a guest context or running the #VMEXIT mechanism to leave a guest context. World switch can also broadly apply to the requisite software effort surrounding VMRUN and #VMEXIT; software effort for some intercepts may be significantly longer than the VMRUN/#VMEXIT portion of the world switch. Several of the optimizations proposed in this chapter attempt to reduce the frequency of world switches. Other optimizations provide techniques to reduce software or processor effort required for performing other virtualization tasks.

For additional information on virtualization and related topics, see Chapter 15, “Secure Virtual Machine,” in the *AMD64 Architecture Programmer’s Manual Volume 2: System Programming* (order# 24593).

This chapter covers the following topics:

Topic	Page
Use Nested Paging	210
VMCB.G_PAT Configuration	211
State Swapping	211
Economizing Interceptions	212
Nested Page Size	213
Shadow Page Size	214
Setting VMCB.TLB_Control	214
TLB Flushes in Shadow Paging	215
Use of Virtual Interrupt VMCB Field	216
Avoid Instruction Fetch for Intercepted (REP) OUTS Instructions	217
Share IOIO and MSR Protection Maps	219
Obey CPUID Results	219
Using Time Sources	220
Paravirtualized Resources	221

12.1 Use Nested Paging

Optimization

❖ Use nested paging instead of shadow paging.

Application

This optimization applies to:

- VMMs

Rationale

To virtualize guests fully, virtual machine monitor (VMM or hypervisor) software must virtualize guests' physical memory mappings without the guests' knowledge. On processors that do not implement nested paging, a method called shadow paging is commonly used for this purpose. But that method is complex to implement efficiently, it is significantly slower than native virtual-to-physical address translation, and performance tuning often requires significant memory to store cached shadow page tables for each guest page table. (There is typically one page table per guest user process.) Shadow paging requires both significant time for the VMM to manage shadow page tables and frequent VMM intervention during guest page faults, guest CR0, CR3, and CR4 accesses, guest INVLPG execution, and guest modifications to page table contents.

In contrast to shadow paging, nested paging requires minimal VMM attention. The CR*x*, INVLPG, and page fault intercepts are unnecessary, and the VMMs need only set up an initial nested page table that maps guest physical addresses to system physical addresses. Each guest requires its own nested page table. A VMM that uses nested paging is significantly less complex and, thus, is easier to validate and verify than a VMM using shadow paging.

A TLB miss under nested paging incurs potentially more memory accesses than a TLB miss under shadow paging, but AMD-V microprocessors that support nested paging employ intelligent caching to minimize the latency of a nested paging TLB miss.

TLB Miss Latency under Nested Paging

TLB entries cache translations from the virtual address to the system physical address for non-virtualized programs and from the guest virtual address to the system physical address for shadow and nested paging. A TLB hit under nested paging performs the same as a TLB hit under shadow paging or in a non-virtualized environment. A TLB miss under nested paging is potentially more expensive than a non-nested TLB miss; nested paging page table walks are accelerated by the CPU's caching of page table information.

12.2 VMCB.G_PAT Configuration

Optimization

❖ Properly configure the guest page attribute table (G_PAT) in the virtual machine control block (VMCB).G_PAT field.

Application

This optimization applies to:

- VMMs using nested paging

Rationale

When nested paging is enabled, the VMCB.G_PAT field is used to virtualize the guest's PAT register. For a description of how the final memory type of a guest page is determined, see section 15.24.8 “Combining Memory Types, MTRRs” in the *AMD64 Architecture Programmer’s Manual Volume 2: System Programming* (order# 24593). For details on the organization and layout of the VMCB, see Appendix B “Layout of VMCB” in the same volume.

Operating systems typically leave the PAT at its default reset value of 0x00070406_00070406, although they are free to change the PAT register's contents. The VMM software should start up guest virtual machines with the same default value. A VMM that leaves the G_PAT value equal to 0x0 will experience significant performance degradation in the guest because all guest memory accesses will be forced to the effective PAT type of uncacheable (UC).

12.3 State Swapping

Optimization

Avoid unnecessary VMSAVE, VMLOAD, STGI, CLGI, and guest GPR and FPR state swapping.

Application

This optimization applies to:

- VMMs for guests in all modes

Rationale

Avoiding unnecessary instructions that would occur on every world switch can reduce the cost of a world switch.

For example, a VMM may need only a small subset of the state swapped by VMSAVE and VMLOAD, so the VMM that expects to return to the same guest can skip VMSAVEing the guest's

state and, instead, leave that guest state active in the CPU. If the VMM needs to use any of the VMSAVE values, such as the task register (TR), the VMM can use the LTR instruction to install the VMM's TR value, while leaving the other guest values intact. Upon returning to the guest, the VMM can VMLOAD the guest or execute the LTR instruction to restore guest state from an artificial TR entry in VMM context. To ensure that the guest VMCB contains the correct TR values, the VMM must intercept the LTR instruction in the guest.

Similar work can be done on other pieces of state represented in VMLOAD and VMSAVE.

VMRUN sets the global interrupt flag (GIF) to 1—equivalent to executing an implicit STGI instruction. Similarly, #VMEXIT clears GIF with an implicit CLGI instruction. A VMM that performs only a minimal amount of work between a #VMEXIT and the next VMRUN may wish to skip executing explicit STGI and CLGI instructions.

VMMs can use methods similar to callee-save to avoid saving and restoring all guest general-purpose registers and floating-point registers if the VMM intends to return to the same guest. This approach is probably most useful for performing lazy floating-point state saves and saving debug registers DR0-DR3.

12.4 Economizing Interceptions

Optimization

Intercept as few MSRs, events, and instructions as possible.

Application

This optimization applies to:

- VMMs

Rationale

To minimize virtualization overhead, VMMs should try to minimize the number of #VMEXITs due to MSR and instruction intercepts.

The VMM should intercept only those MSRs that are critical for system function or security, and which, therefore, must be protected from guest access. The VMM can avoid intercepting MSRs that are frequently used and changed by operating systems, such as GSBASE and KernelGSBase, and all other MSRs that are loaded by VMLOAD, since these MSRs have no system-level side-effects and can be efficiently context switched. VMM writers may evaluate the frequency of reads to specific MSRs that must be intercepted to determine if the following optimization is worthwhile: if the read value is equal to the value the guest expects, then the MSR write may be intercepted while leaving the MSR read unintercepted.

The state that is context switched by AMD-V instructions often does not require intercepts. For example, the IDTR, GDTR, LDTR, and TR read and write intercepts, and PUSHF and POPF intercepts often do not need to be set because VMRUN/#VMEXIT and VMLOAD/VMSAVE appropriately virtualize the related state.

Under nested paging, the paging-related control registers (CR0, CR2, CR3, CR4) and PAT MSR are context switched by VMRUN and #VMEXIT and, thus, often do not need to be intercepted. Similarly, the INVLPG intercept is not necessary under nested paging. In comparison, most shadow paging implementations need to intercept CR0, CR3, and CR4 read and write accesses and the INVLPG instruction, although they can avoid intercepting CR2 accesses.

To avoid the overhead of context switching floating-point state, VMMs can use lazy floating point context switching methods by controlling guest CR0.TS. When the VMM forces CR0.TS to a value other than the value the guest had written, the VMM should intercept CR0 reads and writes in order to properly virtualize CR0.TS.

12.5 Nested Page Size

Optimization

Where possible, use large pages in nested page tables.

Application

This optimization applies to:

- VMMs using nested paging

Rationale

VMMs can realize several performance advantages by using large (2 MB or 1 GB) pages in nested page tables, when it is possible for the VMM to allocate naturally-aligned large pages for portions of guest physical memory images.

The first performance increase comes from reducing multiplicative factors in the cost of TLB misses under nested paging.

Secondly, a common use of large pages is to reduce TLB pressure. For best performance, nested page table entries should be larger than or equal to the size of the corresponding guest page size.

Large pages allow the reduction of the memory footprint used by nested page tables. For each 2-MB large page in a nested page table, an entire 4-KB bottom-level page table becomes unnecessary. For each 1-GB large page, a 4-KB page-directory table becomes unnecessary, as do up to 512 bottom level page tables (each of which occupy 4 KB).

12.6 Shadow Page Size

Optimization

Use large pages where possible in shadow paging.

Application

This optimization applies to:

- VMMs using shadow paging

Rationale

For reasons similar to those enumerated in section 12.5, “Nested Page Size” above, VMMs should attempt to use large pages in shadow page tables for address ranges that the guest maps using large pages. This avoids increasing TLB pressure by the fracturing of large pages into smaller TLB entries and reduces software complexity and memory usage.

When a VMM encounters a 2-MB (or 4-MB or 1-GB) guest page and decides to map it using 4-KB shadow page table entries, the VMM must use memory to store an additional 512 derived entries, or 4 KB, for a shadow page table that does not correspond to any page table in the guest. Additionally, if the guest performs an INVLPG instruction to the guest's 2-MB page, the VMM must clear all 512 of the derived 4-KB entries and must invalidate each 4 KB derived page (in which case it is likely to be more efficient to flush the entire TLB, than to execute 512 INVLPG instructions).

12.7 Setting VMCB.TLB_Control

Optimization

When possible, avoid setting VMCB.TLB_Control to 1.

Application

This optimization applies to:

- VMMs

Rationale

Setting VMCB.TLB_Control to 1 and then VMRUNning that VMCB flushes the entire TLB of all entries, local and global, for all ASID values. Flushing the entire TLB can have minor but noticeable adverse effects on performance by unnecessarily flushing TLB entries from ASIDs other than the current ASID. If capacity misses would not have evicted the other ASIDs' TLB entries, then those TLB entries would be available and useful for avoiding page table walks when the VMM or other guests are executed.

12.8 TLB Flushes in Shadow Paging

Optimization

Change the guest ASID instead of causing TLB flushes in shadow paging.

Application

This optimization applies to:

- VMMs using shadow paging

Rationale

When a VMM is using shadow paging, it must intercept every event in the guest that is defined to cause a TLB flush or TLB line invalidation. The most common case of TLB flush cases is the MOV CR3 instruction. Another frequently-seen instruction, INVLPG, invalidates specified TLB entries. When these instructions are intercepted by an AMD-V processor, the TLB flush or invalidate is suppressed and the VMM is responsible for doing appropriate invalidations after performing appropriate shadow page table manipulations. A simplistic solution is to set VMCB.TLB_CONTROL to 1 to cause a complete TLB flush on the next VMRUN. This simplistic solution may have a negative performance impact due to the complete flushing of global entries and TLB entries for all other ASIDs. Notably, the VMM TLB entries that might otherwise be useful after #VMEXIT are lost. To a second order, the TLB entries of other guests may also have been usefully retained had the TLB_CONTROL field not been set.

In order to retain TLB entries of all other contexts, while properly invalidating the TLB entries for the guest that had executed the TLB flushing instruction, a VMM may retire an ASID from use, rather than flushing the entire TLB. For correctness, the VMM must not allocate that ASID for use again until the TLB has been flushed by setting TLB_CONTROL to 1 in any guest's VMCB and then VMRUNning that guest. Retired ASIDs should be un-retired after executing VMRUN with TLB_CONTROL = 1. (Note that ASIDs and TLB flushes are local to the CPU core on which they are used, so each core may have a different guest running with a given value of ASID.)

For example, if we have guests occupying ASIDs 1, 2, 3, and 4, and the guest with ASID == 3 executes MOV-TO-CR3, then the VMM can mark ASID 3 as unusable, allocate ASID 5, and change

the guest's ASID to 5. This has the side-effect of effectively invalidating all of that guest's global pages.

Note that an intercepted INVLPG instruction can be turned into a shadow page table operation followed by an INVLPGA instruction and does not necessarily require a TLB flush.

12.9 Use of Virtual Interrupt VMCB Field

Optimization

Use the Virtual Interrupt VMCB field instead of event injection when there is only one interrupt pending for the guest.

Application

This optimization applies to:

- VMMs, when VMCB.V_INTR_MASKING == 1 for the guest.

Rationale

VMMs commonly do not allow guests direct access to physical interrupts, choosing instead to virtualize the interrupts using the V_INTR_MASKING and virtual interrupt mechanisms.

AMD-V processors automatically deliver a pending virtual interrupt to the guest when the guest is not masking interrupts due to any of the following:

- Guest EFLAGS.IF == 0
- Guest TPR > priority of pending virtual interrupt
- Guest is in an interrupt shadow

VMMs can avoid the overhead and complexity in software of determining if a guest is ready to take the interrupt by appropriately filling the virtual interrupt fields in the guest VMCB, and can avoid one or more unnecessary world switches. An AMD-V processor automatically clears the V_IRQ valid bit when the interrupt is taken.

By taking these steps, VMMs can provide correct interrupt behavior to the guest while using the smallest possible number of world switches.

12.10 Avoid Instruction Fetch for Intercepted (REP) OUTS Instructions

Optimization

Avoid guest instruction fetch in the common case of VMEXIT_IOIO by inferring the absence of the segment override.

Application

This optimization applies to:

- VMMs

Rationale

EXITINFO1 provides most, but not all, information about the state of an intercepted (REP) OUTS instruction. Specifically, segment override information is not provided, but can be inferred or ignored in the common case. The other IOIO instructions (IN, OUT, and (REP) INS) ignore segment prefixes, so the EXITINFO1 field provides all needed decode information. The state that is provided includes indications as to whether the instruction was an IN, OUT, INS, or OUTS instruction (encoded in the TYPE and STR bits), whether there was a REP prefix, effective address and data sizes, the RIP of the next instruction, and the starting port number.

The OUTS instruction defaults to using the DS segment but obeys segment-override prefixes. Discovering the effective segment in a straightforward manner requires fetching the guest instruction and decoding it.

It may be an expensive operation to fetch and decode guest instruction bytes, though some VMMs may implement fast ways to do so, and these operations can be avoided in most common cases of intercepted (REP) OUTS instructions and can be avoided in all cases for IN, OUT, and (REP) INS.

The VMM can infer the minimum length of the (REP) OUTS instruction by calculating the number of prefixes implied by the EXITINFO1 information. The VMM can then compare the minimum length of the instruction against the known length of the instruction and thus determine if any unaccounted prefixes are present that would require instruction decode.

Example

```
if (exitcode == VMEXIT_IOIO && exitinfo1 & 5 == 4)
      // OUTS instruction: STR and TYPE == 0
    {
     int min_length = 1; // All OUTS instructions are 1-byte opcodes
     int default_addr_size;

     // The default address size is encoded one-hot to match the EXITINFO1
     // encoding: 16-bit mode == 0x1, 32-bit mode == 0x2, 64-bit mode = 0x4
```

```
int default_data_size;

// The default data size is encoded one-hot to match the EXITINFO1 encoding:
// 8 bits == 0x1, 16 bits == 0x2, 32 bits = 0x4

int instr_length = vmcb.exitinfo2 - vmcb.rip;


if (CS.L == 0 && CS.D == 0) {            // 16-bit mode
     default_addr_size = 0x1;
      default_data_size = 0x2; }

else if (CS.L == 0 && CS.D == 1) {       // 32-bit or compatibility mode
          default_addr_size = 0x2;
          default_data_size = 0x4; }

else if (CS.L == 1 && CS.D == 0) {       // 64-bit mode
          default_addr_size = 0x4;
          default_data_size = 0x4; }

// Default data size for this instruction is 32 bits in 64-bit mode.

else error;                           //CS.L == 1 and CS.D == 1 are reserved

if (default_addr_size != (exitinfo1 >> 7) & 7)
     min_length++;                  // infer an address-size override prefix
if (default_data_size != (exitinfo1 >> 4) & 7 && (exitinfo1 >> 4) != 1)
     min_length++;
    // infer a data-size override prefix unless data size is 8-bit
```

```
     if (exitinfo1 & 8)
          min_length++;                    // infer a rep prefix

     if (instr_length != min_length)
         there are more prefixes than can be inferred; must decode instruction
         to determine effective segment
     else
          segment is ES
}
```

12.11 Share IOIO and MSR Protection Maps

Optimization

Share IOIO and MSR protection maps, if possible, to save memory.

Application

This optimization applies to:

- VMMs

Rationale

A VMM running multiple guests typically enforces the same I/O port and MSR restrictions on most or all guests in the system. While a VMM must allocate one VMCB per guest virtual CPU, the VMM can conserve memory by sharing common IOIO and MSR protection maps. These structures can be shared because they are read, but never written, by the CPU. The VMM should be careful about using proper mutual exclusion to handle modifications done to protection maps that are in use on other CPUs.

12.12 Obey CPUID Results

Optimization

❖ Guests should obey CPUID results.

Application

This optimization applies to:

- All programs, operating systems, and libraries

Rationale

Any existing or future operating system, program, or library may be executed in a virtualized environment. A VMM may control the results of CPUID to hide certain capabilities from the guest for various reasons. The VMM may wish to enable migration of a guest from one processor to a processor of a different generation with different features enabled. The VMM provides a set of CPUID results to its guests that represents a common subset of features. That subset may not represent any existing physical processor.

To ensure that programs, libraries, and operating systems work properly in the face of virtualization, all software should obey the results returned by CPUID. The most straightforward way to obey CPUID is to execute CPUID once per program or library initialization and then record the result in an internal data structure. For example, a program may detect the RDTSCP indicator in CPUID and then configure code paths to reflect the presence or absence of RDTSCP. The VMM's control over the RDTSCP CPUID bit will cause the program to exhibit the correct behavior based on whether the VMM wishes to advertise the fact that the current CPU implements the RDTSCP instruction.

This restriction is eased for existing programs and existing methods to detect processor features that already existed at the time AMD-V microprocessors were introduced. For example, before using SSE1 instructions, user programs are required to do a try-catch sequence to determine if the operating system has enabled the XMM registers. This try-catch sequence is still required for SSE1 instructions, but software must adhere to the results of CPUID instruction without a try-catch sequence for detecting new instructions like the SSE3 instruction set.

Future CPU versions may add new instruction encodings to replace formerly undefined encodings. Software should never depend on #UD exceptions from instructions that are currently undefined on any given processor. The UD2 opcode should be used if software wishes to create #UD exceptions.

12.13 Using Time Sources

Optimization

Guests should be careful about using time sources.

Application

This optimization applies to:

- All programs, operating systems, and libraries

Rationale

Programs and operating systems that are not virtualization-aware might assume that the RDTSC instruction, high precision event timers (HPETs), programmable interrupt timers (PITs), and other time sources are monotonically increasing by constant amounts and are usable as a measure of both elapsed time and wall-clock time. When a VMM is present, it necessarily intercepts guest operation

for variable lengths of time, and must make adjustments to the time values read by the guest. These adjustments may break one or more assumptions about time sources. A VMM may choose to adjust the time sources to synchronize them with a wall-clock time so that the guest's time of day measurements are correct, in which case a guest that is continuously monitoring the time will see occasional jumps in the apparent wall-clock time; this may cause fairness problems with the guest's process scheduling. A VMM may choose to adjust the time sources so the guest correctly measures elapsed guest time, which would cause the guest's TSC-based measurement of wall-clock time to be incorrect and may affect time-critical applications such as media playback.

It is unlikely that a guest that is unaware of virtualization will be able to use time sources for all common purposes at the same time. Users should be aware of these pitfalls and understand their implications. As operating systems and programs are written to be aware of virtualization, they should take advantage of any available paravirtualized access to time resources. For their part, VMMs should strive to provide a sufficiently rich and standardized set of paravirtualized timer resources.

12.14 Paravirtualized Resources

Optimization

Guests should detect VMM presence and use paravirtualized resources

Application

This optimization applies to:

- All guests that are aware of virtualization

Rationale

An OS does not implicitly know whether it is a guest or if the OS is running without a VMM present. Some VMMs may support paravirtualization as a means to improve performance or create features. When this is the case, guests should use industry-standard methods to detect VMMs and enumerate the available paravirtualized functions. System resources, such as paging controls in non-nested paging environments, time references, network and video drivers, storage and other device drivers can benefit from paravirtualization.

Appendix A Microarchitecture of AMD Family 10h and 12h Processors

An understanding of the terms *architecture*, *microarchitecture*, and *design implementation* is important when discussing processor design.

The *architecture* consists of the instruction set and those features of a processor that are visible to software programs running on the processor. The architecture determines what software the processor can run. The AMD64 architecture of the AMD Family 10h and 12h processors are compatible with the industry-standard x86 instruction set.

The term *microarchitecture* refers to the design features used to reach the target cost, performance, and functionality goals of the processor. The AMD Family 10h and 12h processors employ a decoupled decode/execution design approach. In other words, decoders and execution units operate essentially independently; the execution core uses a small number of instructions and a simplified circuit design implementation to achieve fast single-cycle execution with fast operating frequencies.

The *design implementation* refers to a particular combination of physical logic and circuit elements that comprise a processor that meets the microarchitecture specifications.

This appendix covers the following topics:

Topic	Page
Key Microarchitecture Features	224
Microarchitecture of AMD Family 10h and 12h Processors	225
Superscalar Processor	225
Processor Block Diagram	225
Processor Cache Operations	228
Branch-Prediction Table	229
Fetch-Decode Unit	230
Sideband Stack Optimizer	230
Instruction Control Unit	230
Translation-Lookaside Buffer	230
Integer Unit	231
Floating-Point Unit	233
Load-Store Unit	234
Write Combining	235
Integrated Memory Controller	235
HyperTransport™ Technology Interface (AMD Family 10h only)	236

A.1 Key Microarchitecture Features

AMD Family 10h and 12h processors include many features designed to improve software performance. The internal design, or *microarchitecture*, of these processors provides the following key features:

- Integrated DDR2 memory controller with memory prefetcher
- Integrated DDR3 memory controller with memory prefetcher (Family 12h)
- 64-Kbyte L1 instruction cache and 64-Kbyte L1 data cache
- On-chip L2 cache
- On-chip L3 cache (Family 10h only)
- Integer hardware divider (Family 12h)
- 32-byte instruction fetch
- Instruction predecode and branch prediction during cache-line fills
- Decoupled decode/execution core
- Three-way AMD64 instruction decoding
- Sideband stack optimizer
- Dynamic scheduling and speculative execution
- Three-way integer execution
- Three-way address generation
- Three-way 128-bit wide floating-point execution
- 3DNow!™ technology, MMX™, SSE, SSE2, SSE3 and SSE4a single-instruction multiple-data (SIMD) instruction extensions
- Advanced Bit Manipulation instructions
- Superforwarding
- Enhanced Prefetch to L1 data cache
- Deep out-of-order integer and floating-point execution
- In 64-bit mode, eight additional XMM registers (for use with SSE, SSE2, SSE3, and SSE4a instructions) and eight additional general-purpose registers (GPRs)
- HyperTransport™ technology

A.2 Microarchitecture of AMD Family 10h and 12h Processors

AMD Family 10h and 12h processors implement the AMD64 instruction set by means of *macro-ops* (the primary units of work managed by the processor) and *micro-ops* (the primitive operations executed in the processor's execution units). These are simple fixed-length operations designed to include direct support for AMD64 instructions and adhere to the high-performance principles of fixed-length encoding, regularized instruction fields, and a large register set. This enhanced microarchitecture enables higher processor core performance and promotes straightforward extensibility for future designs.

A.3 Superscalar Processor

The AMD Family 10h and 12h processors are aggressive, out-of-order, three-way superscalar AMD64 processors. They can fetch, decode, and issue up to three AMD64 instructions per cycle with a centralized instruction control unit (ICU) and two independent instruction schedulers—an integer scheduler and a floating-point scheduler. These two schedulers can simultaneously issue up to nine micro-ops to the three general-purpose integer execution units (ALUs), three address-generation units (AGUs), and three floating-point execution units. The processors move integer instructions through the integer execution pipeline, which consists of the integer scheduler and the ALUs, as shown in Figure 8. Floating-point instructions are handled by the floating-point execution pipeline, which consists of the floating-point scheduler and the floating-point execution units.

A.4 Processor Block Diagram

A block diagram of the AMD Family 10h processor is shown in Figure 8 on page 226.

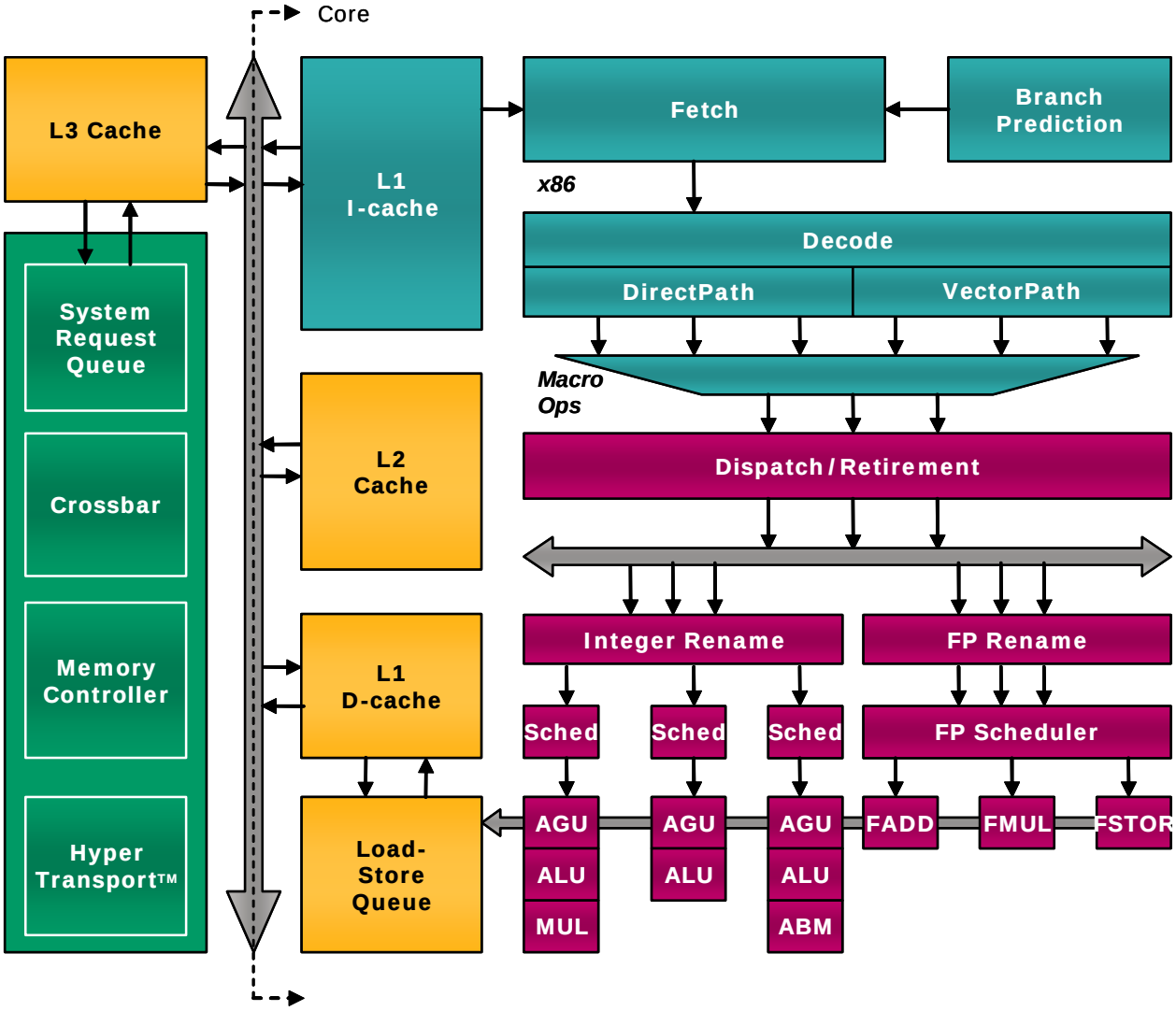


Figure 8. AMD Family 10h Processors Block Diagram

A block diagram of the AMD Family 12h Processor is shown in Figure 9 on page 227.

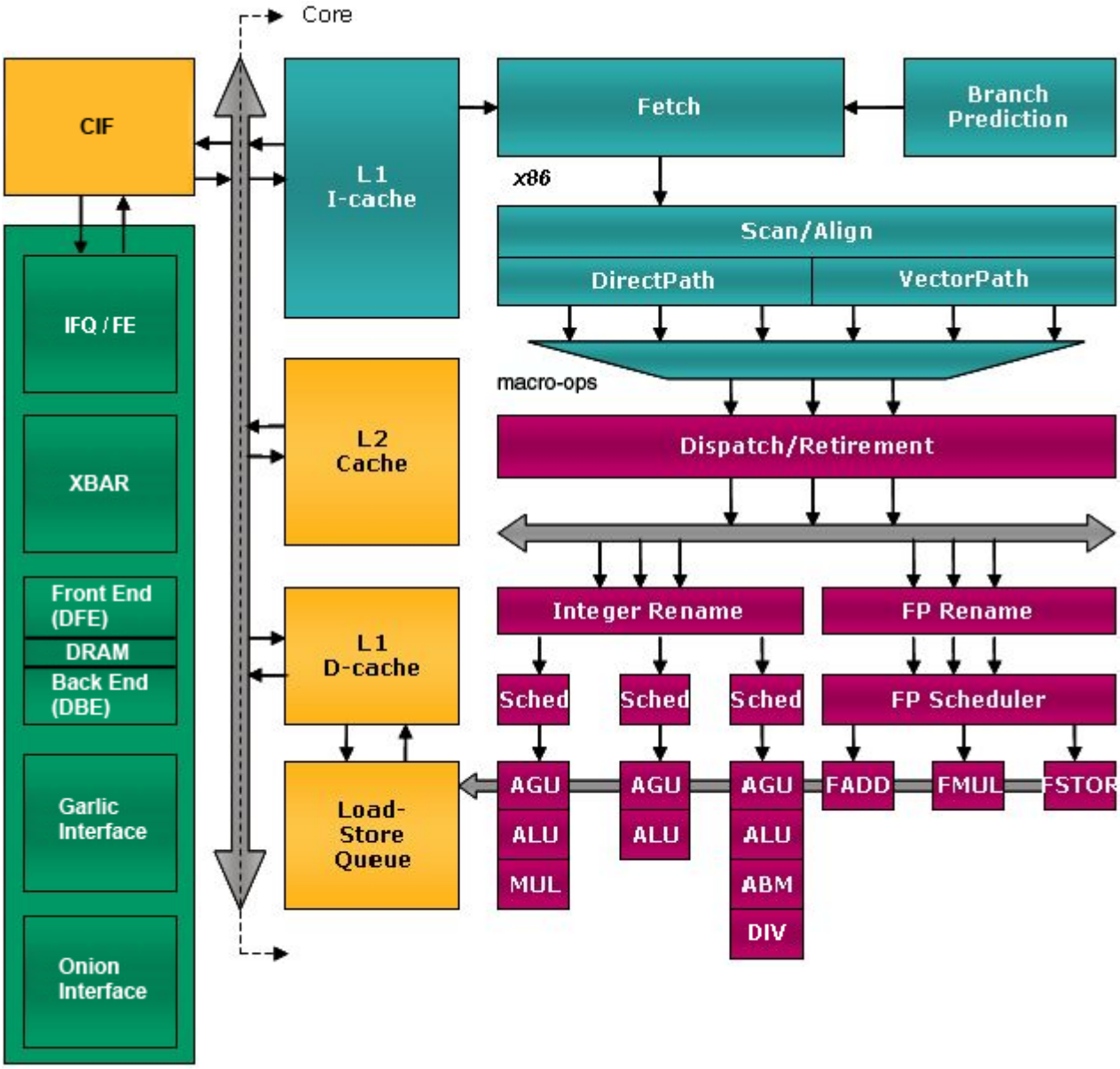


Figure 9. AMD Family 12h Processors Block Diagram

CIF – Synchronizer between CNB and each core; implementing the CCI interface logic.

IFQ/FE – Front End (FE) contains logic that arbitrates between CCI and Onion requests, generates probes, maintains coherency and consistency, and manages the IFQ which is the centralized queue holding CPU and Onion traffic.

XBAR – Read and Write XBAR logic/buses that drives data movements across the CNB.

DRAM – DRAM controller Front End (DFE) and DRAM controller Back End (DBE). DFE selects requests to be sent to DBE and arbitrates between Onion, CCI and Garlic requests. DBE synchronizes between CNB and DRAM and sends the legal sequence of commands to DRAM.

Garlic Interface – Low latency, bi-directional interface between GPU and DBE subblock of CNB.

Onion Interface – Bi-directional interface between the GPU and the IFQ subblock of the CNB.

A.5 Processor Cache Operations

AMD Family 10h and 12h processors use different caches to accelerate instruction execution and data processing.

- L1 instruction cache
- L1 data cache
- L2 cache
- L3 cache (Family 10h only)

A.5.1 L1 Instruction Cache

The out-of-order execution engine of AMD Family 10h and 12h processors contains a 64-Kbyte, 2-way set-associative L1 instruction cache. Each line in this cache is 64 bytes long. Functions associated with the L1 instruction cache are instruction loads, instruction prefetching, instruction predecoding, and branch prediction. Requests that miss in the L1 instruction cache are fetched from the L2 cache or, subsequently, from the L3 cache or system memory using Direct Connect Architecture.

On misses, the L1 instruction cache generates fill requests to a naturally aligned 64-byte line containing the instructions and the next sequential line of 64 bytes (a prefetch). Because code typically exhibits spatial locality, prefetching is an effective technique for avoiding decode stalls. Cache-line replacement is based on a least-recently-used replacement algorithm.

Predecoding begins as the L1 instruction cache is filled. Predecode information is generated and stored alongside the instruction cache. This information is used to help efficiently identify the boundaries between variable length AMD64 instructions.

A.5.2 L1 Data Cache

The AMD Family 10h and 12h processors contain a 64-Kbyte, 2-way set-associative L1 data cache with two 128-bit ports. This cache is a write-allocate and writeback cache that uses a least-recently-used replacement policy. It is divided into eight banks, each 16 bytes wide. In addition, the L1 cache supports the MOESI (Modified, Owner, Exclusive, Shared, and Invalid) cache-coherency protocol and ECC. There is a prefetcher that brings data into the L1 cache to avoid misses. The L1 data cache has a 3-cycle load-to-use latency.

A.5.3 L2 Cache

The AMD Family 10h and 12h processors have one integrated L2 cache per core. This full-speed on-die L2 cache features an exclusive cache architecture. The L2 cache contains only *victim* or *copy-back* cache blocks that are to be written back to the memory subsystem as a result of a conflict miss.

These terms, victim or copy-back, refer to cache blocks that were previously held in the L1 cache but which had to be overwritten (evicted) to make room for newer data. The victim buffer contains data evicted from the L1 cache. The latency of the L2 cache is 9 cycles beyond the L1 cache.

Size and associativity of the AMD Family 10h and 12h processor L2 cache is implementation dependent. See the appropriate BIOS and Kernel Developer's Guide for details.

A.5.4 L3 Cache

The AMD Family 10h processor contains an integrated L3 cache which is dynamically shared between all cores in AMD multi-core processors. The L3 cache is considered a *non-inclusive victim cache* architecture optimized for multi-core AMD processors. Blocks are allocated into the L3 on L2 victim/copy-backs. Requests that hit in the L3 cache can either leave the data in the L3 cache—if it is likely the data is being accessed by multiple cores—or remove the data from the L3 cache (and place it solely in the L1 cache, creating space for other L2 victim/copy-backs), if it is likely the data is only being accessed by a single core. Furthermore, the cache features bandwidth-adaptive policies that optimize latency when requested bandwidth is low, but allows scaling to higher aggregate L3 bandwidth when required (such as in a multi-core environment).

A.6 Branch-Prediction Table

AMD Family 10h and 12h processors predict that a branch is not taken until it is taken once. Then it is predicted that the branch is taken, until it is not taken. Thereafter, the branch prediction table is used.

The fetch logic accesses the branch prediction table in parallel with the L1 instruction cache. The information stored in the branch prediction table is used to predict the direction of branch instructions. When instruction cache lines are evicted to the L2 cache, branch selectors and predecode information are also stored in the L2 cache.

AMD Family 10h and 12h processors employ combinations of a branch target address buffer (BTB), a global history bimodal counter (GHBC) table, and a return address stack (RAS) to predict and accelerate branches. Predicted-taken branches incur only a single-cycle delay to redirect the instruction fetcher to the target instruction. In the event of a misprediction, the minimum penalty is 10 cycles.

The BTB is a 2048-entry table that contains the predicted target address of a branch in each entry. The 16384-entry GHBC table contains 2-bit saturating counters that are used to predict whether a conditional branch is taken. The GHBC table is indexed using the outcome (taken or not taken) of the last conditional branches and the branch address.

AMD Family 10h and 12h processors implement a separate 512- entry target array used to predict indirect branches with multiple dynamic targets.

In addition, the processors implement a 24-entry return address stack to predict return addresses from a near or far call. As calls are fetched, the next return address is pushed onto the return stack. Subsequent returns pop a predicted return address off the top of the stack.

A.7 Fetch-Decode Unit

The fetch-decode unit performs early decoding of AMD64 instructions into macro-ops. AMD Family 10h and 12h processors contain two separate decoders; one to decode DirectPath instructions and one to decode VectorPath instructions. When the target 32-byte instruction window is obtained from the L1 instruction cache, the instruction bytes are examined to determine whether the type of basic decode to take place is DirectPath or VectorPath. The outputs of the early decoders keep all (DirectPath or VectorPath) instructions in program order. Early decoding produces three macro-ops per cycle from either path. The outputs of both decoders are multiplexed together and passed to the next stage in the pipeline, the instruction control unit. Decoding a VectorPath instruction may prevent the simultaneous decoding of a DirectPath instruction.

A.8 Sideband Stack Optimizer

The *Sideband Stack Optimizer* tracks the stack-pointer value. This allows the processor to execute in parallel any set of one or more instructions that implicitly or explicitly reference the stack-pointer. “Stack Operations” on page 59 discusses the Sideband Stack Optimizer in greater detail.

A.9 Instruction Control Unit

The *instruction control unit* (ICU) is the control center for the AMD Family 10h and 12h processors. It controls the centralized in-flight reorder buffer, the integer scheduler, and the floating-point scheduler. In turn, the ICU is responsible for the following functions: macro-op dispatch, macro-op retirement, register and flag dependency resolution and renaming, execution resource management, interrupts, exceptions, and branch mispredictions.

The instruction control unit takes the three macro-ops that are produced during each cycle from the early decoders and places them in a centralized, fixed-issue reorder buffer. For AMD Family 12h processors, this buffer is organized into 28 lines of three macro-ops each. The reorder buffer allows the instruction control unit to track and monitor up to 84 in-flight macro-ops (whether integer or floating-point) for maximum instruction throughput. The instruction control unit can simultaneously dispatch multiple macro-ops from the reorder buffer to both the integer and floating-point schedulers for final decode, issue, and execution as micro-ops.

A.10 Translation-Lookaside Buffer

A *translation-lookaside buffer* (TLB) holds the most-recently-used page mapping information. It assists and accelerates the translation of virtual addresses to physical addresses.

The AMD Family 10h and 12h processors utilize a two-level TLB structure.

A.10.1 L1 Instruction TLB Specifications

The processors contains a fully-associative L1 instruction TLB with 32 4-Kbyte page entries and 16 2-Mbyte page entries. 4-Mbyte pages require two 2-Mbyte entries; thus, the number of entries available for 4-Mbyte pages is one half the number of 2-Mbyte page entries.

A.10.2 L1 Data TLB Specifications

The processors contains a fully-associative L1 data TLB with 48 entries for 4-Kbyte, 2-Mbyte, and 1-Gbyte pages. 4-Mbyte pages require two 2-Mbyte entries; thus, the number of entries available for 4-Mbyte pages is one half the number of 2-Mbyte page entries.

A.10.3 L2 Instruction TLB Specifications

The processors contains a 4-way set-associative L2 instruction TLB with 512 4-Kbyte page entries.

A.10.4 L2 Data TLB Specifications

The AMD Family 10h and 12h processors contain an L2 data TLB with 512 (1024 for 12h processors) 4-Kbyte page entries (4-way set-associative), 128 2-Mbyte page entries (2-way set-associative), and 16 1-Gbyte page entries (8-way set-associative). 4-Mbyte pages require two 2-Mbyte entries; thus, the number of entries available for 4-Mbyte pages is one half the number of 2-Mbyte page entries.

A.11 Integer Unit

The *integer unit* consists of two components, the *integer scheduler*, which feeds the integer execution pipes, and the *integer execution* unit, which carries out several types of operations discussed below.

A.11.1 Integer Scheduler

The integer scheduler is based on a three-wide queuing system (also known as a reservation station) that feeds three integer execution positions or pipes. The reservation stations are eight entries deep, for a total queuing system of 24 integer macro-ops (30 for 12h processors). Each reservation station divides the macro-ops into integer and address generation micro-ops, as required.

A.11.2 Integer Execution Unit

The integer execution pipeline consists of three identical pipes (0, 1, and 2). Each integer pipe consists of an arithmetic-logic unit (ALU) and an address generation unit (AGU). The integer execution pipeline is organized to match the three macro-op dispatch pipes in the ICU as shown in Figure 10.

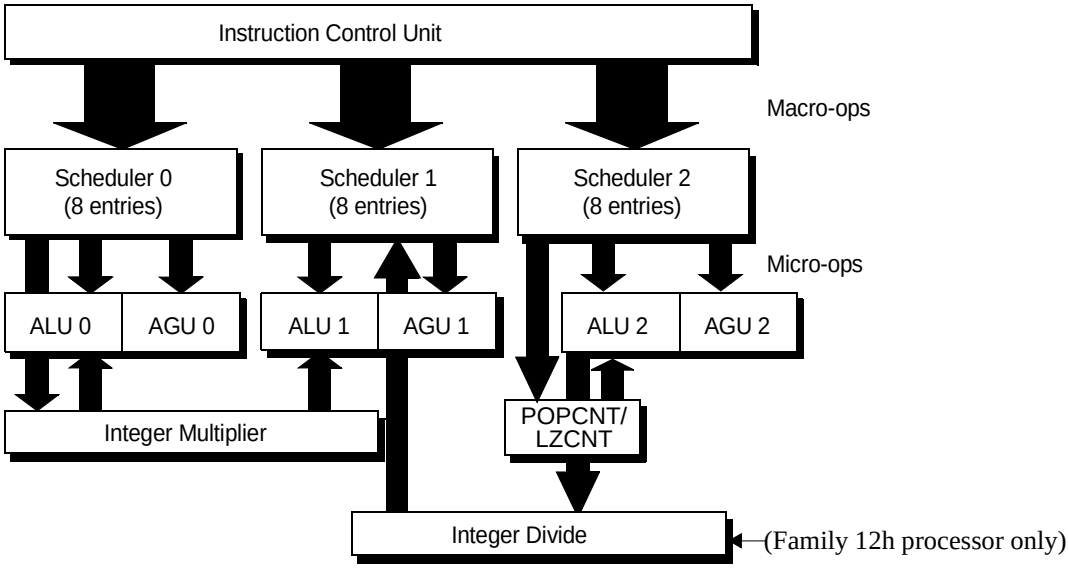


Figure 10. Integer Execution Pipeline

Macro-ops are broken down into micro-ops in the schedulers. Micro-ops are executed when their operands are available, either from the register file or result buses. Micro-ops from a single operation can execute out-of-order. In addition, a particular integer pipe can execute two micro-ops from different macro-ops (one in the ALU and one in the AGU) at the same time. (See Figure 10.)

Each of the three ALUs performs general purpose logic functions, arithmetic functions, conditional functions, divide step functions, status flag multiplexing, and branch resolutions. The AGUs calculate the logical addresses for loads, stores, and LEAs. A load and store unit reads and writes data to and from the L1 data cache. The integer scheduler sends a completion status to the ICU when the outstanding micro-ops for a given macro-op are executed. (For more information on the LSU, see section A.13 on page 234.)

All integer operations can be handled within any of the three ALUs with the exception of multiplication, division, LZCNT, and POPCNT operations. Multiplication is handled by a pipelined multiplier that is attached to the pipeline at pipe 0, as shown in Figure 10 on page 232. Multiplication operations always issue to integer pipe 0, and the issue logic creates result bus bubbles for the multiplier in integer pipes 0 and 1 by preventing non-multiply micro-ops from issuing at the appropriate time. Integer divisions are always issued to pipe 2 and issue logic creates result bus bubbles for the divider in the integer pipes 1 and 2 by preventing non-division operations from issuing at the appropriate time. The LZCNT and POPCNT operations are handled in a pipelined unit attached to pipe 2, as shown in Figure 10 on page 232. The LZCNT/POPCNT operations always issue to integer pipe 2, and the issue logic creates a result bus bubble in integer pipe 2 by preventing non-LZCNT/POPCNT operations from issuing at the appropriate time.

A.12 Floating-Point Unit

The *floating-point unit* consists of two components, the *floating-point scheduler*, which performs several complex functions prior to actually feeding into the *floating-point execution unit*, which carries out several types of operations discussed below.

A.12.1 Floating-Point Scheduler

The floating-point logic of the AMD Family 10h and 12h processors are a high-performance, fully pipelined, superscalar, out-of-order execution unit. It is capable of accepting three macro-ops per cycle from any mixture of the following types of instructions:

- x87 floating-point
- 3DNow! technology
- MMX
- SSE
- SSE2
- SSE3
- SSE4a

The floating-point scheduler handles register renaming and has a dedicated 42-entry scheduler buffer organized as 14 lines of three macro-ops each. It also performs data superforwarding, micro-op issue, and out-of-order execution. The floating-point scheduler communicates with the ICU to retire a macro-op, to manage results of *COMI* and FP-to-INT movement and conversion instructions using a 64-bit-wide FP-to-INT bus, and to back out results from a branch misprediction.

Superforwarding is a performance optimization. It allows faster scheduling of a floating point operation having a dependency on a register when that register is waiting to be filled by a pure load from memory. Instead of waiting for the first instruction to write its load-data to the register and then waiting for the second instruction to read it, the load-data can be provided directly to the dependent instruction, much like regular forwarding between FPU-only operations. The result from the load is said to be "superforwarded" to the floating-point operation. In the following example, the FADD can be scheduled to execute as soon as the load operation fetches its data rather than having to wait and read it out of the register file.

```
fld    [somefloat]        ;Load a floating point
                           ;value from memory into ST(0)
fadd   st(0),st(1)        ;The data from the load will be
                           ;forwarded directly to this instruction,
                           ;no need to read the register file
```

A.12.2 Floating-Point Execution Unit

The floating-point execution unit (FPU) has its own out-of-order execution control and datapath. The FPU handles all register operations for x87 instructions, all 3DNow! technology operations, all MMX operations, and all SSE, SSE2, SSE3, and SSE4a operations. The FPU consists of a stack renaming unit, a register renaming unit, a scheduler, a register file, and execution units that are each capable of computing and delivering results of up to 128 bits per cycle. Figure 11 shows a block diagram of the dataflow through the FPU.

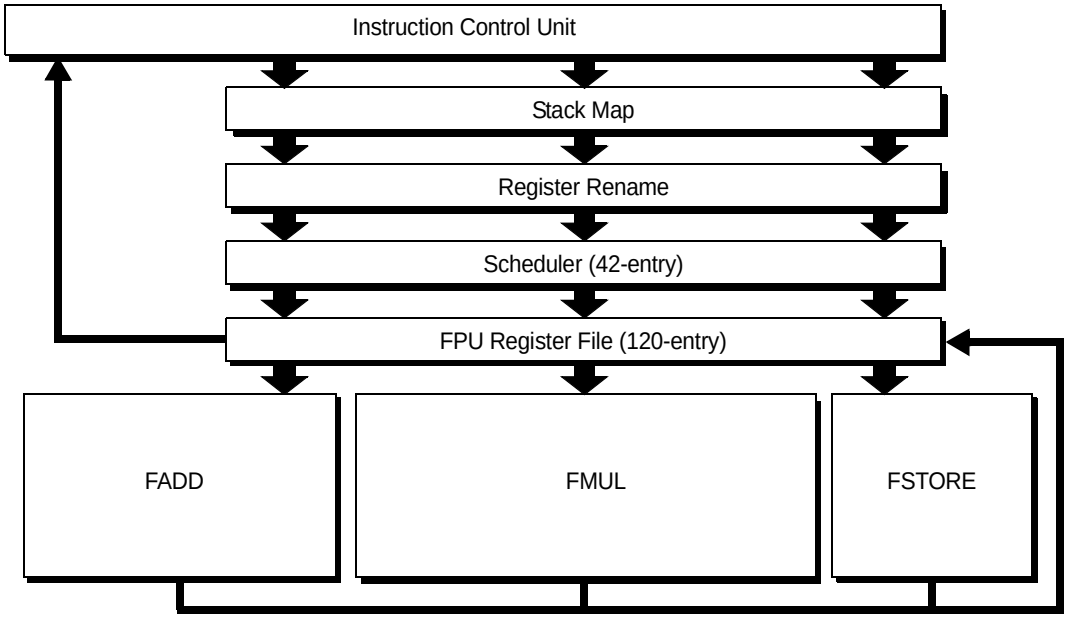


Figure 11. Floating-Point Unit

As shown in Figure 11, the floating-point logic uses three separate execution positions or pipes (FADD, FMUL, and FSTORE). Details on which instructions can use which pipes are specified in Appendix C.

A.13 Load-Store Unit

The L1 data cache and load-store unit (LSU) are shown in Figure 11. The L1 data cache can support two 128-bit loads or two 64-bit store writes per cycle or a mix of those. The LSU consists of two queues—LS1 and LS2. LS1 can issue two L1 cache operations (loads or store tag checks) per cycle. It can issue load operations out-of-order, subject to certain dependency restrictions. LS2 effectively holds requests that missed in the L1 cache after they probe out of LS1. Store writes are done exclusively from LS2. 128-bit stores are specially handled in that they take two LS2 entries, and the store writes are performed as two 64-bit writes. Finally, the LSU helps ensure that the architectural load and store ordering rules are preserved (a requirement for AMD64 architecture compatibility).

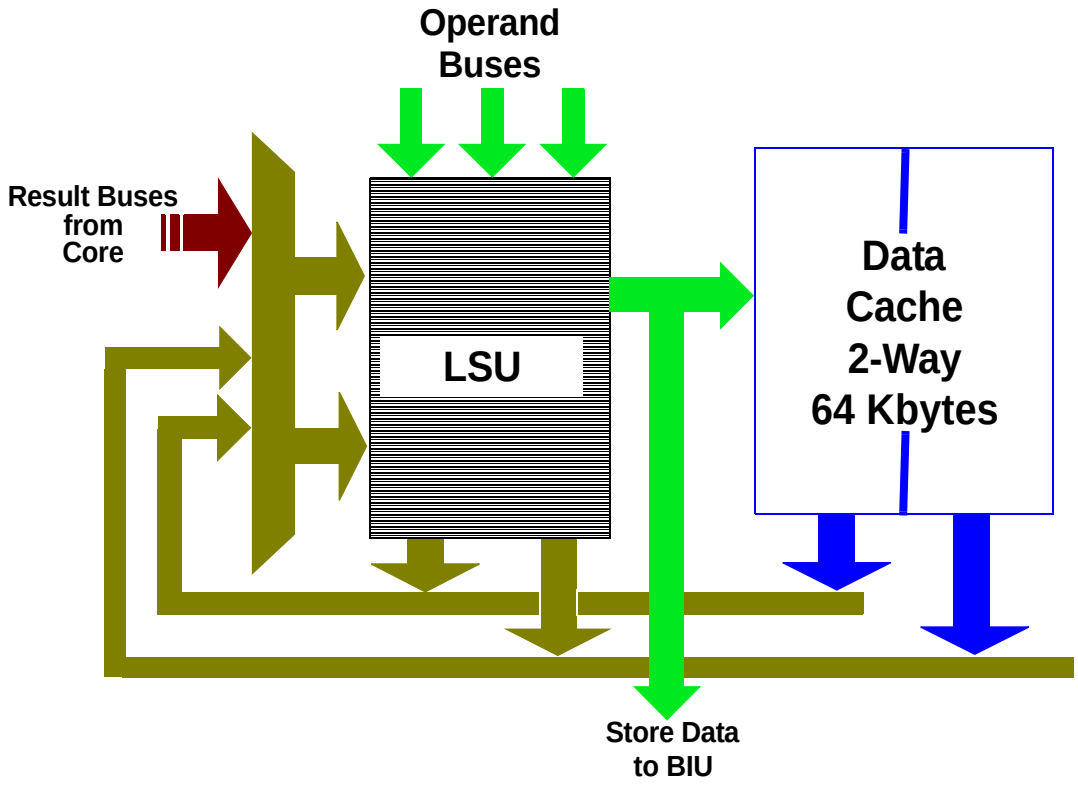


Figure 12. Load-Store Unit

A.14 Write Combining

AMD Family 10h and 12h processors provide four write-combining data buffers that allows merging four streams, each to a different 64B cache line. There are eight write-combining address buffers so each core can support eight outstanding requests to the CNB. For details, see Appendix B “Implementation of Write-Combining” on page 239.

A.15 Integrated Memory Controller

AMD Family 10h processors provide integrated low-latency, high-bandwidth DDR2 and DDR3 memory controllers. AMD Family 12h processors provide integrated low-latency, high-bandwidth DDR3 memory controllers.

The memory controller supports:

- DRAM chips that are 4 (Family 10h only), 8, and 16 bits wide within a DIMM.
- Interleaving memory among Chip Selects and among DRAM channels.
- ECC checking with double-bit detection and single-bit correction. (AMD Family 10h only)

- Dual-independent 64-bit channel and single 128-bit channel operation. (AMD Family 12h only supports dual, independent 64-bit channels).
- Optimized scheduling algorithms and dynamic selection of open/close page policy to improve latency and achieved bandwidth, particularly for interleaved streams of read and write DRAM accesses.
- Optimized scheduling algorithm to minimize latency and maximize bandwidth for interleaved streams of CPU/Onion and Garlic accesses. (AMD Family 12h supports Onion and Garlic interfaces).
- A data prefetcher.

Prefetched data is held in the memory controller itself and is not speculatively filled into the L1, L2, or L3 caches (L3 cache applies to Family 10h only). This prefetcher is able to capture both positive and negative stride values (both unit and non-unit) at the cache-line size granularity, as well as sequences of strides. It can also simultaneously track up to 8 different strides or sequences of strides. (This applies to the data prefetcher of the AMD Family 12h only).

For specifications on a certain processor's memory controller, see the data sheet for that processor. For information on how to program the memory controller, see the *BIOS and Kernel Developer's Guide for AMD Family 10h Processors*, order# 31116.

A.16 HyperTransport™ Technology Interface (AMD Family 10h only)

HyperTransport technology is a scalable, high-speed, low-latency, point-to-point, packetized link that:

- Enables high data transfer rates.
- Simplifies connectivity by replacing legacy buses and bridges.
- Reduces latencies and bottlenecks within systems.

When compared with traditional technologies, HyperTransport technology allows much faster data-transfer rates. For more information on HyperTransport technology, see the *HyperTransport I/O Link Specification*, available at www.hypertransport.org.

On AMD Family 10h processors, HyperTransport technology provides the link to I/O devices. Some processor models—for example, those designed for use in multiprocessing systems—also utilize HyperTransport technology to connect to other processors. See the *BIOS and Kernel Developer's Guide* for your particular processor for details concerning HyperTransport technology implementation details.

In addition to supporting previous HyperTransport interfaces, AMD Family 10h processors support a newer version of the HyperTransport standard: HyperTransport3. HyperTransport3 increases the aggregate link bandwidth to a maximum of 20.8 Gbyte/s (16-bit link). HyperTransport3 also adds

HyperTransport Retry which improves RAS by allowing detection and retransmission of packets corrupted in transit.

Additional features in the AMD Family 10h HyperTransport implementation may include:

- HyperTransport link bandwidth balancing, allowing multiple HyperTransport links to be teamed to carry coherent traffic.
- HyperTransport Link Splitting, which allowing a single 16-bit link to be split into two 8-bit links.

These features allow for further optimized platform designs that are capable of increasing system bandwidth and reducing latency.

A.16.1 HyperTransport Assist (AMD Family 10h only)

Six core and higher multisocket-capable AMD family 10h processors incorporate HyperTransport assist technology (also referred to in some documents as probe filtering). HyperTransport assist functionality may or may not be enabled by default on a specific platform implementation. However, the BIOS can enable HyperTransport assist on these platforms, if it is not enabled by default.

HyperTransport assist reduces the effective latency of memory access in multi-node systems by changing the coherence protocol from a broadcast style to a directory style. In cases requiring many memory accesses, in particular to local memory—as is common in NUMA-optimized applications—the probe and response latency required for maintaining coherence takes longer than the DRAM access. In these cases, HyperTransport assist can remove many of the probes and responses associated with DRAM access, thus decreasing effective latency to access system memory.

HyperTransport assist also increases the total coherent fabric bandwidth within the system by removing much probe and response traffic from the coherent HyperTransport links. It also streamlines probe and response handling throughout the L1/L2/L3 caches and elsewhere in the microarchitecture, which can lead to additional bandwidth improvements in systems with multiple processing nodes.

HyperTransport assist is enabled by partitioning the L3 cache physical storage into a section used as traditional (CPU-side) L3 cache, and a separate physical section for directory storage which is inaccessible to the CPUs. In effect, from the perspective of CPUs, systems with HyperTransport assist enabled have a smaller L3 cache. Typically, 1MB of L3 cache is reserved for use by HyperTransport assist technology. Thus, some amount of L3 capacity is traded for reduced latency on cache refills. While the benefit of this tradeoff can be workload-dependant, it is almost universally a win on larger (4+ node) systems. If a platform runs a specific workload, it may be worth evaluating performance with and without HyperTransport assist.

Enabling HyperTransport assist allows for performance benefits beyond the NUMA optimizations suggested in Appendix D, “Tools and APIs for AMD Family 10h and 12h ccNUMA Multiprocessor Systems,” on page 289.

For information on HyperTransport assist implementation details for a specific processor, see the BIOS and Kernel Developer's Guide for AMD Family 10h Processors, order #31116 (where it goes by the alternate name of probe filter).

Appendix B Implementation of Write-Combining

This appendix describes the memory write-combining feature implemented in AMD Family 10h and 12h processors. Write-combining is the merging of multiple memory write operations that target locations within the address range of a write buffer.

AMD Family 10h and 12h processors support the memory type range register (MTRR) and the page attribute table (PAT) extensions, which allow software to define ranges of memory as either writeback (WB), write-protected (WP), writethrough (WT), uncacheable (UC), or write-combining (WC).

Defining the memory type for a range of memory as WC or WT allows the processor to conditionally combine data from multiple write operations that are addressed within this range into a merge buffer. Merging multiple write operations into a single write operation reduces processor bus utilization and processor stalls. Write combining buffers are also used for streaming store instructions such as MOVNTQ and MOVNTI. See “Use of Streaming Instructions” on page 84.

This appendix covers the following topics:

Topic	Page
Write-Combining Definitions and Abbreviations	239
Programming Details	240
Write-Combining Operations	240
Sending Write-Buffer Data to the System	241
Write Combining to MMI/O Devices that Support Write Chaining	241

B.1 Write-Combining Definitions and Abbreviations

This appendix uses the following definitions and abbreviations:

- MTRR—Memory type range register
- PAT—Page attribute table
- UC—Uncacheable memory type
- WC—Write-combining memory type
- WT—Writethrough memory type
- WP—Write-protected memory type
- WB—Writeback memory type

B.2 Programming Details

Write-combining regions are controlled by the MTRRs and PAT extensions. Write-combining should be enabled for the appropriate memory ranges. (For more information on the MTRRs and the PAT extensions, see the *AMD64 Architecture Programmer's Manual, Volume 2*, order# 24593, and the *BIOS and Kernel Developer's Guide for AMD Family 10h Processors*, order# 31116.)

B.3 Write-Combining Operations

To improve system performance, AMD Family 10h and 12h processors aggressively combine multiple memory-write operations of any data size that address locations within a 64-byte write buffer that is aligned to a cache-line boundary. The processor continues to combine writes to this buffer without writing the data to the system, as long as certain rules apply (see Table 11 for more information). The data sizes can be bytes, words, doublewords, or quadwords.

- WC memory type writes can be combined in any order up to a full 64-byte write buffer.
- All other memory types for stores that go through the write buffer (UC, WP, WT and WB) cannot be combined except when the WB memory type is over-ridden for streaming store instructions such as the MOVNTQ and MOVNTI instructions, etc. These instructions use the write buffers and will be write-combined in the same way as address spaces mapped by the MTTR registers and PAT extensions. When WC is used for streaming store instructions, then the buffers are subject to the same flushing events as write-combined address spaces.

Combining continues until interrupted by one of the conditions listed in Table 11. When combining is interrupted, one or more bus commands are issued to the system for that write buffer and all older write buffers, even if they are not full, as described in "Sending Write-Buffer Data to the System" on page 241.

Table 11. Write-Combining Completion Events

Event	Comment
I/O Read or Write	Any IN/INS or OUT/OUTS instruction closes combining. The implied memory type for all IN/OUT instructions is UC, which cannot be combined.
Serializing instructions	Any serializing instruction closes combining. These instructions include: MOVCRx, MOVDRx, WRMSR, INVD, INVLPG, WBINVD, LGDT, LLDT, LIDT, LTR, CPUID, IRET, RSM, INIT, and HALT.
Flushing instructions	Any flush instruction causes the WC to complete.
Locks	Any instruction or processor operation that requires a cache or bus lock closes write-combining before starting the lock. Writes within a lock can be combined.
Uncacheable Read	A UC read closes write-combining. A WC read closes combining only if a cache block address match occurs between the WC read and a write in the write buffer.

Table 11. Write-Combining Completion Events (Continued)

Event	Comment
Different memory type	When a store hits on a write buffer that has been written to earlier with a different memory type than that store, the buffer is closed and flushed.
Buffer full	Write-combining is closed if all 64 bytes of the write buffer are valid.
TLB AD bit set	Write-combining is closed whenever a TLB reload sets the accessed [A] or dirty [D] bits of a PDE or PTE.

B.4 Sending Write-Buffer Data to the System

Maximum throughput is achieved by write combining when all quadwords or doublewords are valid and the AMD Family 10h and 12h processors can use one efficient 64-byte memory write instead of multiple 8-byte memory writes.

B.5 Write Combining to MMI/O Devices that Support Write Chaining

AMD Family 10h and 12h processors support four write-combining buffers. Although the number of buffers available for write combining depends on the specific CPU revision, current designs provide as many as four write buffers for WC memory mapped I/O address spaces. Although the core can combine WC data for up to four cache lines, it can sustain up to eight pending Write requests to the CNB.

These same buffers are used for streaming store instructions. The number of write-buffers determines how many independent linear 64-byte streams of WC data the CPU can simultaneously buffer.

Having multiple write-combining buffers that can combine independent WC streams has implications on data throughput rates (bandwidth), especially when data is written by the CPU to WC memory mapped I/O devices, residing on the AGP, PCI, PCI-X® and PCI Express® buses including:

- Memory Mapped I/O registers—command FIFO, etc.
- Memory Mapped I/O apertures—windows to which the CPU uses programmed I/O to send data to a hardware device
- Sequential block of 2D/3D graphic engine registers written using programmed I/O
- Video memory residing on the graphics accelerator—frame buffer, render buffers, textures, etc.

HyperTransport™ Tunnels and Write Chaining (AMD Family 10h only)

HyperTransport™ tunnels are HyperTransport-to-bus bridges. Many HyperTransport tunnels use a hardware optimization feature called write-chaining. In write-chaining, the tunnel device buffers and combines separate HyperTransport packets of data sent by the CPU, creating one large burst on the

underlying bus when the data is received by the tunnel in sequential address order. Using larger bursts results in better throughput since bus efficiency is increased. This is because bus arbitration overhead is lower: only one address/attribute phase is issued per burst in the PCI-X case, and one address/command phase is issued for the AGP Fast Writes case.

For reasons cited in the preceding paragraph, to utilize hardware write chaining efficiently, software should flush the CPU write-combining buffer in sequential linear address order, any time a target GB/s hardware device is capable of receiving large bursts of CPU write data.

Software should be aware that on AMD64 processors that have multiple write-combining buffers, events that flush the write-combining buffers (see Table 11 on page 240) do so in the order that the streams were opened. For example, if the CPU writes to the WC space in the 64-byte buffer at the highest address first (say, address 40h), followed by a write to a lower 64-byte buffer (for example, address 00h), the CPU first sends the highest addressed 64-byte buffer by HyperTransport to the tunnel, followed by the second (lower address) 64-byte buffer. Since the addressing is not sequential the tunnel device will not chain both 64-byte WC buffers and must issue two separate transactions on the target bus.

If the buffers in this example were targeted for AGP fast writes, issuing two fast write transactions (rather than issuing one fast write transaction) would reduce the bandwidth (data throughput) by one-third.

Optimizations

Adhere to the following guidelines to ensure that AMD Family 10h and 12h processors issue WC buffers in sequential address order:

- When practical, shadow the data structure in memory (rather than writing the actual WC buffer in MMI/O space), prior to copying the structure to WC MMI/O space. This will also ensure that the write-combining buffers are not emptied prematurely by external events (such as a UC read—perhaps issued by another device driver thread or a hardware interrupt, etc.). Shadowing also ensures that writes that occur to different cache lines in the structure do not send out the WC buffers, since the number of WC buffers that can be open at one time is CPU implementation dependent.
- When ready to update the actual WC MMI/O address space, copy the shadowed structure from memory to MMI/O, from the lowest address 64-byte block upward. To do the copy, use discrete loads and stores for up to 64 bytes of data. Use a loop of discrete loads and stores for up to 4KB of data. When copying the shadowed structure, if possible, group together the discrete loads in order to minimize interleaving them with the discrete stores. That way, the number of read-write and write-read turnaround delays on the DRAM will be minimized and the data throughput of the copy operation will be maximized. Use REP MOVS instructions for up to 32KB of data. To do discrete loads use assembly language, or, if available, compiler intrinsic functions available (`__movsb()`, `__movsw()`, `__movsd()`), etc. (For more information, see “Memory and String Routines” on page 92.)

- In general, using these methods to do the copy will exhibit less overhead in a data movement function than calling a `memcpy()` LIBC function, which is usually optimized for copying larger blocks of memory.

Appendix C Instruction Latencies

This appendix provides a listing of AMD64 instructions, decode types, and execution latencies. For more information on these instructions, see the *AMD64 Architecture Programmer's Manual, Volumes 3, 4, and 5* (order# 24594, 26568, and 26569).

The instruction entries in this appendix are grouped into categories as follows and are presented within each category in alphabetical order by mnemonic:

Topic	Page
Understanding Instruction Entries	246
General Purpose and Integer Instruction Latencies	250
System Instruction Latencies	261
128-Bit Media Instruction Latencies	265
64-Bit Media Instruction Latencies	279
x87 Floating-Point Instruction Latencies	284

C.1 Understanding Instruction Entries

To use the information in this appendix effectively, you need to understand how the entry for an instruction is organized and how to interpret certain items.

Example: Instruction Entry

The entry for an instruction begins with its syntax. Subsequent columns provide additional information about the instruction.

Syntax	Decode Type	FPU Pipe(s)	Latency	Throughput	Notes
MOVAPS *mem*, *reg*	DirectPath Double	FSTORE	2	1/1	4

Parts of the Instruction Entry

Columns in the latency tables are defined as follows. Not all categories are relevant to all instruction sets. Thus, only the decode type and latency are relevant to general purpose integer instructions, while SSE*x* latency tables use all six categories.

Category	Description
Syntax	Shows the syntax for the instruction—the permitted arrangement of its parts. Items in italics are placeholders for operands that you must provide. For information on how to interpret the placeholders, see “Interpreting Placeholders” on page 247
Decode type	Shows the method that the processor uses to decode the instruction—DirectPath Single, DirectPath Double, or VectorPath.
FPU Pipes	Lists the possible floating-point unit (FPU) pipelines available for use by any particular DirectPath or Double decoded operation. (See below.)
Latency	Shows the static execution latency for the instruction. For details on how to interpret the latency information, see “Interpreting Latencies” on page 248.
Throughput	This value indicates the maximum theoretical rate of execution of that instruction. For example, a value of 1/2 means that one such instruction executes every two clocks, or two such instructions in four clocks and so on. A value of 3/1 indicates that three such instructions can be executed every clock, but fewer than three such instructions would still take one clock.
Notes	Specifies clarifying information

Decode Type

The ★ and ▲ symbols indicate a one step change in the decode type hierarchy on AMD Family 10h and 12h processors as compared to the decode type of the identical instruction on 8th generation AMD processors. The decode type hierarchy, from simplest to most complex is:

DirectPath Single↔DirectPath Double↔VectorPath.

Instructions having more complex decode types decompose into more micro-operations, each of which consumes important system resources. A positive two-step decode-type change (from VectorPath to DirectPath) is indicated by ★★; a negative two-step change in decode-type (from DirectPath to VectorPath) is indicated by ▲▲.

FPU Pipes

The entries for floating-point, MMX, SSE, SSE2, SSE3, and SSE4a instructions have an additional column [FPU Pipe(s)] that lists the possible floating-point unit (FPU) pipelines available for use by any particular DirectPath Single or DirectPath Double decoded operation. The floating-point add pipe is represented by FADD, the floating point multiply pipe is represented by FMUL and the floating-point store pipe is represented by FSTORE. A '/' between two pipe names indicates that an instruction can use either of the two pipes. An '&' between pipe names indicates that both pipes are required. An entry such as "FADD & FSTORE" thus indicates that both an FADD pipe and an FSTORE pipe are used; "(FADD/FMUL) & FSTORE" indicates that either an FADD or an FMUL pipe is used in addition to a (required) FSTORE pipe.

Interpreting Placeholders

The Syntax column for an instruction entry shows the mnemonic for the instruction followed by any operands. Items in italics are placeholders for operands that you must provide. A placeholder indicates the size and type of operand that is allowed.

This operand	Is a placeholder for
reg	A general-purpose register
mmreg	An MMX™ register
xmmreg	An XMM, SSE, SSE2, SSE3, SSE4a register
ST(*i*)	X87 stack register
mem	A memory location
imm	An *imm*ediate value
disp	A memory *disp*lacement or offset
x/y	Operand type x or y
x (mem)	Operand type x or mem (used only for media instructions)
Note: *Operands with numbers indicate operand sizes, for example mem32/64 indicates that this operand can either be a 32-bit or a 64-bit memory location. When sizes are not indicated, the information in the entry is identical for any legal operand size. Please consult the AMD64 Architecture Programmer's Manual Volumes 3–5 to determine the legal operand sizes for a given instruction type.*	

In many if not most cases, an instruction takes more that one operand. When an instruction takes two register operands, as in `ADD reg1, reg2`, the first register (*reg1*) is the *destination* operand (or register) and the second register (`reg2`) is the *source* operand (or register). In the latency table that follows, numeric suffixes are used to discriminate between operands in all such cases (*i.e.*, `xmmreg2`, `mmreg1`, *etc.*). A few instructions take three operands; the same conventions for designating operands apply in these cases as well, as in:

```
EXTRQ xmmreg, imm1, imm2
CMPSS xmmreg1, xmmreg2 (mem), imm
```

Interpreting Latencies

The Latency column for an instruction entry shows the static execution latency for the instruction. The static execution latency is the number of clock cycles it takes to execute the serially dependent sequence of micro-ops that comprise the instruction.

The latencies in this appendix are estimates and are subject to change. They assume that:

- The instruction is an L1-cache hit that has already been fetched and decoded, with the operations loaded into the scheduler.
- Memory operands are in the L1 data cache.
- There is no contention for execution resources or load-store unit resources.

Each latency in the table denotes the typical execution time of the instruction when run in isolation on a processor with any referenced memory locations already in the L1 cache. For real programs executed on this highly aggressive superscalar family of processors, multiple instructions can execute simultaneously; therefore, the effective latency for any given instruction's execution may be overlapped with the latency of other instructions executing in parallel. An example of this effect can be seen for an SSE load-compute instruction like `ADDPD reg, mem`, which effectively adds 2 cycles of latency (6 cycles total) versus `ADDPD reg, reg` (4 cycles) when run in isolation. In a real program, however, the load portion of the instruction often occurs in parallel with earlier work, effectively hiding the extra 2 cycles from the critical execution path. There are also other cases of additional latencies that may be incurred in a real program that are not described in the latency table, such as delays caused by L1 cache misses or contention for execution or load-store unit resources.

The following formats are used to indicate the static execution latency:

Table 12. Latency Formats

Latency format	Description	Example
x	The latency is the indicated value.	3
x/y/z	The latency differs according to the size of the operands. The values *x*, *y*, and *z* are the 16-, 32-, and 64-bit latencies, respectively. When used in latency values for x87 instructions, this notation is used to indicate latencies for different precision control modes (single precision/double precision/extended precision).	26/42/74
x (y)	The latency depends on whether the particular form of the instruction takes a memory operand or a register operand. The latency of the register-operand form of the instruction is specified first; the latency of the memory-operand form is given in parentheses.	2 (4)

VectorPath Instruction Latencies

In the following tables, certain VectorPath instruction latencies may not precisely reflect all versions of the processor.

C.2 General Purpose and Integer Instruction Latencies

The latency table for general purpose and integer instructions gives the decode type and latency corresponding to each instruction mnemonic. For more detailed information on the operation of a particular general purpose integer instruction, as well as encoding information, see the *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*, order# 24594.

Table 13. General Purpose and Integer Instruction Latencies

Syntax	Decode Type[1]	Latency	Notes
AAA	VectorPath	5	
AAD	VectorPath	5	
AAM	VectorPath	14	
AAM	VectorPath	15 (Family 12h)	
AAS	VectorPath	5	
ADC *reg*, *reg/imm*	DirectPath Single	1	
ADC *mem*, *reg/imm*	DirectPath Single	4	
ADC *reg*, *mem*	DirectPath Single	4	
ADD *reg*, *reg/imm*	DirectPath Single	1	
ADD *mem*, *reg/imm*	DirectPath Single	4	
ADD *reg*, *mem*	DirectPath Single	4	
AND *reg*, *reg/imm*	DirectPath Single	1	
AND *mem*, *reg/imm*	DirectPath Single	4	
AND *reg*, *mem*	DirectPath Single	4	

Note:

1. *For interpretation of special symbols, see "Decode Type" on page 247.*
2. *See "Repeated String Instructions" on page 128.*
3. *For information on calculating latencies for the DIV/IDIV instructions, see "DIV/IDIV Latencies" on page 144.*
4. *The NOP instruction does not consume any execution resources.*
5. *This operation is restricted to scheduling in pipe 0.*
6. *This operation is restricted to scheduling in pipe 2.*
7. *These instructions use the processor's write-combining resources.*
8. *LEA instructions have a latency of 1 when there are two source operands (as in the case of the base + index form LEA EAX, [EDX+EDI]). Forms with a scale or more than two source operands have a latency of 2 (LEA EAX, [EBX+EBX*8]).*
9. *For information on Family 12h division instructions, see first paragraph in "Rationale" on page 121.*

Table 13. General Purpose and Integer Instruction Latencies (Continued)

Syntax	Decode Type[1]	Latency	Notes
BOUND *reg32*, *mem64*	VectorPath	6	
BSF *reg*, *reg*	VectorPath	4	
BSF *reg*, *mem*	VectorPath	7	
BSR *reg*, *reg*	VectorPath	4	
BSR *reg*, *mem*	VectorPath	7	
BSWAP *reg*	DirectPath Single	1	
BT *reg*, *reg/imm*	DirectPath Single	1	
BT *mem*, *imm*	DirectPath Single	4	
BT *mem*, *reg*	VectorPath	7	
BTC *reg*, *reg/imm*	DirectPath Double	2	
BTC *mem*, *imm*	VectorPath	5	
BTC *mem*, *reg*	VectorPath	8	
BTR *reg*, *reg/imm*	DirectPath Double	2	
BTR *mem*, *imm*	VectorPath	5	
BTR *mem*, *reg*	VectorPath	8	
BTS *reg*, *reg/imm*	DirectPath Double	2	
BTS *mem*, *imm*	VectorPath	5	
BTS *mem*, *reg*	VectorPath	8	
CALL *disp* (near)	DirectPath Double★	3	
CALL *reg* (near)	DirectPath Double★	3	
CALL *mem* (near)	VectorPath	4	
CBW/CWDE/CDQE	DirectPath Single	1	

Note:

1. *For interpretation of special symbols, see “Decode Type” on page 247.*
2. *See “Repeated String Instructions” on page 128.*
3. *For information on calculating latencies for the DIV/IDIV instructions, see “DIV/IDIV Latencies” on page 144.*
4. *The NOP instruction does not consume any execution resources.*
5. *This operation is restricted to scheduling in pipe 0.*
6. *This operation is restricted to scheduling in pipe 2.*
7. *These instructions use the processor's write-combining resources.*
8. *LEA instructions have a latency of 1 when there are two source operands (as in the case of the base + index form LEA EAX, [EDX+EDI]). Forms with a scale or more than two source operands have a latency of 2 (LEA EAX, [EBX+EBX*8]).*
9. *For information on Family 12h division instructions, see first paragraph in “Rationale” on page 121.*

Table 13. General Purpose and Integer Instruction Latencies (Continued)

Syntax	Decode Type[1]	Latency	Notes
CWD/CDQ/CQO	DirectPath Single	1	
CLC	DirectPath Single	1	
CLD	DirectPath Single	1	
CMC	DirectPath Single	1	
CMOV*cc reg*, *reg*	DirectPath Single	1	
CMOV*cc reg*, *mem*	DirectPath Single	4	
CMP *reg*, *reg*/*imm*	DirectPath Single	1	
CMP *mem*, *reg*/*imm*	DirectPath Single	4	
CMP *reg*, *mem*	DirectPath Single	4	
CMPS/CMPSB/CMPSW/CMPSD/CMPSQ	VectorPath	6	2
CMPXCHG *reg*, *reg*	VectorPath	3	
CMPXCHG *mem8*, *reg8*	VectorPath	6	
CMPXCHG *mem16*/*32*/*64*, *reg16*/*32*/*64*	VectorPath	5	
CMPXCHG8B *mem64*	VectorPath	10	
CMPXCHG16B *mem128*	VectorPath	11	
CPUID fn0x0	VectorPath	41	
CPUID fn0x1	VectorPath	128	
CPUID fn0x2	VectorPath	37	
CPUID fn 1	VectorPath	564 (Family 12h)	
CPUID fn 2	VectorPath	38 (Family 12h)	
DAA	VectorPath	7	
DAS	VectorPath	7	

Note:

1. *For interpretation of special symbols, see “Decode Type” on page 247.*
2. *See “Repeated String Instructions” on page 128.*
3. *For information on calculating latencies for the DIV/IDIV instructions, see “DIV/IDIV Latencies” on page 144.*
4. *The NOP instruction does not consume any execution resources.*
5. *This operation is restricted to scheduling in pipe 0.*
6. *This operation is restricted to scheduling in pipe 2.*
7. *These instructions use the processor's write-combining resources.*
8. *LEA instructions have a latency of 1 when there are two source operands (as in the case of the base + index form LEA EAX, [EDX+EDI]). Forms with a scale or more than two source operands have a latency of 2 (LEA EAX, [EBX+EBX*8]).*
9. *For information on Family 12h division instructions, see first paragraph in “Rationale” on page 121.*

Table 13. General Purpose and Integer Instruction Latencies (Continued)

Syntax	Decode Type[1]	Latency	Notes
DEC *reg*	DirectPath Single	1	
DEC *mem*	DirectPath Single	4	
DIV reg/mem	VectorPath		3
DIV 8+ ceil(num_quot_bits/2) for quotient, +2 for remainder (Family 12h)	VectorPath		9
ENTER *imm32*, 0/1/2	VectorPath	14/17/19	
IDIV reg/mem	VectorPath		3
IDIV 8+ ceil(num_quot_bits/2) for quotient, +2 for remainder (Family 12h)	VectorPath		9
IMUL *reg8*	DirectPath Single	3	5
IMUL *reg16*	VectorPath	4	5
IMUL *reg16*, *imm16*	VectorPath	4	5
IMUL *reg16*, *mem16*	DirectPath Single	6	5
IMUL *reg16*, *mem16*, *imm*	VectorPath	7	5
IMUL *reg16*, *reg16*	DirectPath Single	3	5
IMUL *reg16*, *reg16*, *imm*	VectorPath	4	5
IMUL *reg32*	DirectPath Double	3	5
IMUL *reg32*, *imm32*	DirectPath Single	3	5
IMUL *reg32*, *mem32*	DirectPath Single	6	5
IMUL *reg32*, *mem32*, *imm*	VectorPath	7	5
IMUL *reg32*, *reg32*	DirectPath Single	3	5
IMUL *reg32*, *reg32*, *imm*	DirectPath Single	3	5
IMUL *reg64*	DirectPath Double	5	5

Note:

1. *For interpretation of special symbols, see “Decode Type” on page 247.*
2. *See “Repeated String Instructions” on page 128.*
3. *For information on calculating latencies for the DIV/IDIV instructions, see “DIV/IDIV Latencies” on page 144.*
4. *The NOP instruction does not consume any execution resources.*
5. *This operation is restricted to scheduling in pipe 0.*
6. *This operation is restricted to scheduling in pipe 2.*
7. *These instructions use the processor's write-combining resources.*
8. *LEA instructions have a latency of 1 when there are two source operands (as in the case of the base + index form LEA EAX, [EDX+EDI]). Forms with a scale or more than two source operands have a latency of 2 (LEA EAX, [EBX+EBX*8]).*
9. *For information on Family 12h division instructions, see first paragraph in “Rationale” on page 121.*

Table 13. General Purpose and Integer Instruction Latencies (Continued)

Syntax	Decode Type[1]	Latency	Notes
IMUL *reg64*, *imm32*	DirectPath Single	4	5
IMUL *reg64*, *mem64*	DirectPath Single	7	5
IMUL *reg64*, *mem64*, *imm*	VectorPath	8	5
IMUL *reg64*, *reg64*	DirectPath Single	4	5
IMUL *reg64*, *reg64*, *imm32*	DirectPath Single	4	5
IMUL *mem8*	DirectPath Single	6	5
IMUL *mem16*	VectorPath	7	5
IMUL *mem32*	DirectPath Double	6	5
IMUL *mem64*	DirectPath Double	8	5
INC *reg*	DirectPath Single	1	
INC *mem*	DirectPath Single	4	
J*cc* disp	DirectPath Single	1	
JCXZ/JECXZ/JRCXZ *disp*	DirectPath Double	2	
JMP reg (near)	DirectPath Single	1	
JMP *disp* (near)	DirectPath Single	1	
JMP *mem* (near)	DirectPath Single	4	
JMP *disp* (far, no call gate)	VectorPath	21	
JMP *mem* (far, no call gate)	VectorPath	22	
LAHF	VectorPath	3	
LEA *reg16*, *mem*	VectorPath	3	
LEA *reg32*/*64*, *mem*	DirectPath Single	1/2	8
LEAVE	DirectPath Double★	3	

Note:

1. *For interpretation of special symbols, see “Decode Type” on page 247.*
2. *See “Repeated String Instructions” on page 128.*
3. *For information on calculating latencies for the DIV/IDIV instructions, see “DIV/IDIV Latencies” on page 144.*
4. *The NOP instruction does not consume any execution resources.*
5. *This operation is restricted to scheduling in pipe 0.*
6. *This operation is restricted to scheduling in pipe 2.*
7. *These instructions use the processor's write-combining resources.*
8. *LEA instructions have a latency of 1 when there are two source operands (as in the case of the base + index form LEA EAX, [EDX+EDI]). Forms with a scale or more than two source operands have a latency of 2 (LEA EAX, [EBX+EBX*8]).*
9. *For information on Family 12h division instructions, see first paragraph in “Rationale” on page 121.*

Table 13. General Purpose and Integer Instruction Latencies (Continued)

Syntax	Decode Type[1]	Latency	Notes
LODS/LODSB	VectorPath	5	2
LODS/LODSW	VectorPath	5	2
LODS/LODSD	VectorPath	4	2
LOOP/LOOP*cc pm32*	VectorPath	8	
LOOP/LOOP*cc pm64*	VectorPath	7	
LZCNT *reg*, *reg*	DirectPath Single	2	6
LZCNT *reg*, *mem*	DirectPath Single	5	6
MOV *reg*, *reg*	DirectPath Single	1	
MOV *reg*, *mem8/16*	DirectPath Single	4	
MOV reg, mem32/63	DirectPath Single	3	
MOV *mem*, *reg/imm*	DirectPath Single	3	
MOV *mem16*, FS	DirectPath Double	4	
MOV *mem32*, SS	DirectPath Double	4	
MOV *mem32*, DS	DirectPath Double	4	
MOV *reg32*, SS	DirectPath Single	4	
MOV *reg32*, DS	DirectPath Single	4	
MOV *reg32*, FS	DirectPath Single	3	
MOV *reg64*, FS	DirectPath Single	3	
MOV SS, *mem32*	VectorPath	26	
MOV SS, *reg32*	VectorPath	10	
MOV DS, *mem32*	VectorPath	10	
MOV DS, *reg32*	VectorPath	8	

Note:

1. *For interpretation of special symbols, see “Decode Type” on page 247.*
2. *See “Repeated String Instructions” on page 128.*
3. *For information on calculating latencies for the DIV/IDIV instructions, see “DIV/IDIV Latencies” on page 144.*
4. *The NOP instruction does not consume any execution resources.*
5. *This operation is restricted to scheduling in pipe 0.*
6. *This operation is restricted to scheduling in pipe 2.*
7. *These instructions use the processor's write-combining resources.*
8. *LEA instructions have a latency of 1 when there are two source operands (as in the case of the base + index form LEA EAX, [EDX+EDI]). Forms with a scale or more than two source operands have a latency of 2 (LEA EAX, [EBX+EBX*8]).*
9. *For information on Family 12h division instructions, see first paragraph in “Rationale” on page 121.*

Table 13. General Purpose and Integer Instruction Latencies (Continued)

Syntax	Decode Type[1]	Latency	Notes
MOV FS, *mem16*	VectorPath	10	
MOV FS, *reg32*	VectorPath	8	
MOV FS, *reg64*	VectorPath	8	
MOVNTI *mem*, *reg*	DirectPath Single		7
MOVS/MOVSB/MOVSW/MOVSD/MOVSQ[1]	VectorPath	5	2
MOVSX *reg*, *reg*	DirectPath Single	1	
MOVSX *reg*, *mem*	DirectPath Single	4	
MOVSXD *reg*, *reg*	DirectPath Single	1	
MOVSXD *reg*, *mem*	DirectPath Single	4	
MOVZX *reg*, *reg*	DirectPath Single	1	
MOVZX *reg*, *mem*	DirectPath Single	4	
MUL *reg8*	DirectPath Single	3	5
MUL *reg16*	VectorPath	4	5
MUL *reg32*	DirectPath Double	3	5
MUL *reg64*	DirectPath Double	5	5
MUL *mem8*	DirectPath Single	6	5
MUL *mem16*	VectorPath	7	5
MUL *mem32*	DirectPath Double	6	5
MUL *mem64*	DirectPath Double	8	5
NEG *reg*	DirectPath Single	1	
NEG *mem*	DirectPath Single	4	
NOP	DirectPath Single	~0	4

Note:

1. *For interpretation of special symbols, see “Decode Type” on page 247.*
2. *See “Repeated String Instructions” on page 128.*
3. *For information on calculating latencies for the DIV/IDIV instructions, see “DIV/IDIV Latencies” on page 144.*
4. *The NOP instruction does not consume any execution resources.*
5. *This operation is restricted to scheduling in pipe 0.*
6. *This operation is restricted to scheduling in pipe 2.*
7. *These instructions use the processor's write-combining resources.*
8. *LEA instructions have a latency of 1 when there are two source operands (as in the case of the base + index form LEA EAX, [EDX+EDI]). Forms with a scale or more than two source operands have a latency of 2 (LEA EAX, [EBX+EBX*8]).*
9. *For information on Family 12h division instructions, see first paragraph in “Rationale” on page 121.*

Table 13. General Purpose and Integer Instruction Latencies (Continued)

Syntax	Decode Type[1]	Latency	Notes
NOT *reg*	DirectPath Single	1	
NOT *mem*	DirectPath Single	4	
OR *reg*, *reg*/*imm*	DirectPath Single	1	
OR *mem*, *reg*/*imm*	DirectPath Single	4	
OR *reg*, *mem*	DirectPath Single	4	
POP *reg16*	DirectPath Double★	4	
POP *reg32*/*64*	DirectPath Single★★	3	
POP *mem*	VectorPath	3	
POP DS/ES/FS/GS	VectorPath	10	
POP SS	VectorPath	26	
POPA/POPAD	VectorPath	6	
POPCNT *reg*, *reg*	DirectPath Single	2	6
POPCNT *reg*, *mem*	DirectPath Single	5	6
POPF/POPFD/POPFQ	VectorPath	15	
PUSH *reg/imm*	DirectPath Single	3	
PUSH *mem*	DirectPath Double	3	
PUSH CS/DS/ES/FS/GS/SS	DirectPath Double★	3	
PUSHA/PUSHAD	VectorPath	6	
PUSHF/PUSHFD/PUSHFQ	VectorPath	–	
RCL *reg*, 1	DirectPath Single	1	
RCL *reg*, *imm*	VectorPath	7	
RCL *reg*, CL	VectorPath	6	

Note:

1. *For interpretation of special symbols, see “Decode Type” on page 247.*
2. *See “Repeated String Instructions” on page 128.*
3. *For information on calculating latencies for the DIV/IDIV instructions, see “DIV/IDIV Latencies” on page 144.*
4. *The NOP instruction does not consume any execution resources.*
5. *This operation is restricted to scheduling in pipe 0.*
6. *This operation is restricted to scheduling in pipe 2.*
7. *These instructions use the processor's write-combining resources.*
8. *LEA instructions have a latency of 1 when there are two source operands (as in the case of the base + index form LEA EAX, [EDX+EDI]). Forms with a scale or more than two source operands have a latency of 2 (LEA EAX, [EBX+EBX*8]).*
9. *For information on Family 12h division instructions, see first paragraph in “Rationale” on page 121.*

Table 13. General Purpose and Integer Instruction Latencies (Continued)

Syntax	Decode Type[1]	Latency	Notes
RCL *mem*, 1	DirectPath Single	4	
RCL *mem*, *imm*	VectorPath	7	
RCL *mem*, CL	VectorPath	7	
RCR *reg*, 1	DirectPath Single	1	
RCR *reg*, *imm*	VectorPath	5	
RCR *reg*, CL	VectorPath	4	
RCR *mem*, 1	DirectPath Single	4	
RCR *mem*, *imm*	VectorPath	6	
RCR *mem*, CL	VectorPath	6	
RET	DirectPath Single★	4	
RET *imm16*	DirectPath Double★	4	
ROL *reg*, 1/CL/*imm*	DirectPath Single	1	
ROL *mem*, 1/CL/*imm*	DirectPath Single	4	
ROR *reg*, 1/CL/*imm*	DirectPath Single	1	
ROR *mem*, 1/CL/*imm*	DirectPath Single	4	
SAHF	DirectPath Single	1	
SAL/SHL *reg*, 1/CL/*imm*	DirectPath Single	1	
SAL/SHL *mem*, 1/CL/*imm*	DirectPath Single	4	
SAR *reg*, 1/CL/*imm*	DirectPath Single	1	
SAR *mem*, 1/CL/*imm*	DirectPath Single	4	
SBB *reg*, *reg*/*imm*	DirectPath Single	1	
SBB *mem*, *reg*/*imm*	DirectPath Single	4	

Note:

1. *For interpretation of special symbols, see “Decode Type” on page 247.*
2. *See “Repeated String Instructions” on page 128.*
3. *For information on calculating latencies for the DIV/IDIV instructions, see “DIV/IDIV Latencies” on page 144.*
4. *The NOP instruction does not consume any execution resources.*
5. *This operation is restricted to scheduling in pipe 0.*
6. *This operation is restricted to scheduling in pipe 2.*
7. *These instructions use the processor's write-combining resources.*
8. *LEA instructions have a latency of 1 when there are two source operands (as in the case of the base + index form LEA EAX, [EDX+EDI]). Forms with a scale or more than two source operands have a latency of 2 (LEA EAX, [EBX+EBX*8]).*
9. *For information on Family 12h division instructions, see first paragraph in “Rationale” on page 121.*

Table 13. General Purpose and Integer Instruction Latencies (Continued)

Syntax	Decode Type[1]	Latency	Notes
SBB *reg*, *mem*	DirectPath Single	4	
SCAS/SCASB/SCASW/SCASD/SCASQ	VectorPath	4	2
SET*cc reg*	DirectPath Single	1	
SET*cc mem*	DirectPath Single	3	
SHLD *reg*, *reg*, CL/*imm*	VectorPath	4	
SHLD *mem*, *reg*, CL/*imm*	VectorPath	6	
SHR *reg*, 1/CL/*imm*	DirectPath Single	1	
SHR *mem*, 1/CL/*imm*	DirectPath Single	4	
SHRD *reg*, *reg*, CL/*imm*	VectorPath	4	
SHRD *mem*, *reg*, CL/*imm*	VectorPath	6	
STC	DirectPath Single	1	
STD	DirectPath Double	2	
STOS/STOSB/STOSW/STOSD/STOSQ	VectorPath	4	2
SUB *reg*, *reg*/*imm*	DirectPath Single	1	
SUB *mem*, *reg*/*imm*	DirectPath Single	4	
SUB *reg*, *mem*	DirectPath Single	4	
TEST *reg*, *reg*/*imm*	DirectPath Single	1	
TEST *mem*, *reg*/*imm*	DirectPath Single	4	
XADD *reg, reg*	VectorPath	2	
XADD *mem*, *reg*	VectorPath	5	
XCHG *reg8*, *reg8*	VectorPath	2	
XCHG *reg16*/*32*/*64*, *reg16*/*32*/*64*	DirectPath Double★	1	

Note:

1. *For interpretation of special symbols, see “Decode Type” on page 247.*
2. *See “Repeated String Instructions” on page 128.*
3. *For information on calculating latencies for the DIV/IDIV instructions, see “DIV/IDIV Latencies” on page 144.*
4. *The NOP instruction does not consume any execution resources.*
5. *This operation is restricted to scheduling in pipe 0.*
6. *This operation is restricted to scheduling in pipe 2.*
7. *These instructions use the processor's write-combining resources.*
8. *LEA instructions have a latency of 1 when there are two source operands (as in the case of the base + index form LEA EAX, [EDX+EDI]). Forms with a scale or more than two source operands have a latency of 2 (LEA EAX, [EBX+EBX*8]).*
9. *For information on Family 12h division instructions, see first paragraph in “Rationale” on page 121.*

Table 13. General Purpose and Integer Instruction Latencies (Continued)

Syntax	Decode Type[1]	Latency	Notes
XCHG *reg8*, *mem8*	VectorPath	16	
XCHG *reg16*, *mem16*	DirectPath Double★	16	
XCHG *reg32/64*, *mem32/64*	DirectPath Double★	15	
XCHG *mem8*, *reg8*	VectorPath	16	
XCHG *mem16*, *reg16*	DirectPath Double★	16	
XCHG *mem32/64*, *reg32/64*	DirectPath Double★	15	
XLAT/XLATB	VectorPath	5	
XOR *reg*, *reg/imm*	DirectPath Single	1	
XOR *mem*, *reg/imm*	DirectPath Single	4	
XOR *reg*, *mem*	DirectPath Single	4	

Note:

1. *For interpretation of special symbols, see “Decode Type” on page 247.*
2. *See “Repeated String Instructions” on page 128.*
3. *For information on calculating latencies for the DIV/IDIV instructions, see “DIV/IDIV Latencies” on page 144.*
4. *The NOP instruction does not consume any execution resources.*
5. *This operation is restricted to scheduling in pipe 0.*
6. *This operation is restricted to scheduling in pipe 2.*
7. *These instructions use the processor's write-combining resources.*
8. *LEA instructions have a latency of 1 when there are two source operands (as in the case of the base + index form LEA EAX, [EDX+EDI]). Forms with a scale or more than two source operands have a latency of 2 (LEA EAX, [EBX+EBX*8]).*
9. *For information on Family 12h division instructions, see first paragraph in “Rationale” on page 121.*

C.3 System Instruction Latencies

The latency table for system instructions gives the decode type and latency corresponding to each instruction mnemonic. For more detailed information on the operation of a particular system instruction, as well as encoding information, see the *AMD64 Architecture Programmer’s Manual Volume 3: General-Purpose and System Instructions*, order# 24594 .

Table 14. System Instruction Latencies

Syntax	Decode Type[1]	Latency	Notes
ARPL *reg16*, *reg16*	VectorPath	13	
ARPL *mem16*, *reg16*	VectorPath	18	
CLI	VectorPath	3	
CLTS	VectorPath	11	
INVLPG mem8	DirectPath	74/95	2
IRETQ	VectorPath	80	
LAR *reg*, *reg*	VectorPath	15	
LAR *reg*, *mem*	VectorPath	17	
LGDT *mem32*	VectorPath	36	
LIDT *mem32*	VectorPath	36	
LLDT *reg32*	VectorPath	32	
LLDT *mem32*	VectorPath	33	
LMSW *reg*	VectorPath	14	
LMSW *mem*	VectorPath	15	
LSL *reg*, *reg16*	VectorPath	15	
LSL *reg*, reg32/64	VectorPath	14	
LSL *reg*, *mem16*	VectorPath	17	
LSL *reg*, *mem32*/*64*	VectorPath	16	
MONITOR	DirectPath		5
MOV CR0, *reg32*	VectorPath	60	6
MOV CR0, *reg64*	VectorPath	61	6
MOV CR2, *reg32*	VectorPath	40	

Note:

1. *For interpretation of special symbols, see “Decode Type” on page 247.*
2. *Values correspond to 64-bit mode/32-bit mode.*
3. *45 core clocks + 16 Northbridge clocks.*
4. *The latency of RDMSR and WRMSR are dependent on the particular machine register being accessed.*
5. *MONITOR and MWAIT are only used in multi-threaded environments; thus, the latency of the MONITOR/MWAIT idiom is highly variable. The latencies provided here are estimations of a lower bound. For MWAIT this encompasses the entire latency of the instruction (both the time before and after an incoming store to the monitored region).*
6. *The latency of this instruction is variable and depends on which register bits change.*

Table 14. System Instruction Latencies (Continued)

Syntax	Decode Type[1]	Latency	Notes
MOV CR4, *reg*32/64	VectorPath	55	6
MOV CR8, *reg32*	VectorPath	33	
MOV DR0–3, *reg32*	VectorPath	63	
MOV DR0–3, *reg32*	VectorPath	64 (Family 12h)	
MOV DR0, *reg64*	VectorPath	65	
MOV DR0, *reg64*	VectorPath	66 (Family 12h)	
MOV DR6–7, *reg32*	VectorPath	52	
MOV DR6–7, *reg32*	VectorPath	55 (Family 12h)	
MOV DR6, *reg64*	VectorPath	54	
MOV DR6, *reg64*	VectorPath	57 (Family 12h)	
MOV *reg32*, CR0	VectorPath	14	
MOV *reg32*, CR2	VectorPath	16	
MOV *reg32*, CR3	VectorPath	14	
MOV *reg32*, CR4	VectorPath	14	
MOV *reg32*, CR8	VectorPath	16	
MOV *reg32*, DR0–3	VectorPath	16	
MOV *reg32*, DR0–3	VectorPath	22 (Family 12h)	
MOV *reg32*, DR6–7	VectorPath	16	
MOV *reg32*, DR6–7	VectorPath	22 (Family 12h)	
MOV *reg64*, CR0	VectorPath	16	
MOV *reg64*, CR3	VectorPath	16	
MOV *reg64*, CR4	VectorPath	16	
MOV *reg64*, DR0	VectorPath	16	

Note:

1. *For interpretation of special symbols, see “Decode Type” on page 247.*
2. *Values correspond to 64-bit mode/32-bit mode.*
3. *45 core clocks + 16 Northbridge clocks.*
4. *The latency of RDMSR and WRMSR are dependent on the particular machine register being accessed.*
5. *MONITOR and MWAIT are only used in multi-threaded environments; thus, the latency of the MONITOR/MWAIT idiom is highly variable. The latencies provided here are estimations of a lower bound. For MWAIT this encompasses the entire latency of the instruction (both the time before and after an incoming store to the monitored region).*
6. *The latency of this instruction is variable and depends on which register bits change.*

Table 14. System Instruction Latencies (Continued)

Syntax	Decode Type[1]	Latency	Notes
MOV *reg64*, DR0	VectorPath	22 (Family 12h)	
MOV *reg64*, DR6	VectorPath	16	
MOV *reg64*, DR6	VectorPath	22 (Family 12h)	
MWAIT	DirectPath		5
RDMSR APIC base	VectorPath	68	
RDMSR FS base	VectorPath	38	
RDMSR GS base	VectorPath	38	
RDMSR	VectorPath		4
RDPMC	VectorPath	9	
RDTSC	VectorPath	45 + 16	3
RDTSCP	VectorPath	45 + 16	3
SGDT *mem*	VectorPath	7	
SIDT *mem*	VectorPath	7	
SLDT *reg/mem*	VectorPath	5	
SMSW *reg/mem*	VectorPath	5	
STI	VectorPath	3	
STR *reg/mem*	VectorPath	5	
SWAPGS	VectorPath	18	
VERR/VERW *reg16*	VectorPath	8	
VERR/VERW *mem16*	VectorPath	10	
WRMSR APIC base	VectorPath	119	
WRMSR FS base	VectorPath	59	
WRMSR GS base	VectorPath	59	

Note:

1. *For interpretation of special symbols, see "Decode Type" on page 247.*
2. *Values correspond to 64-bit mode/32-bit mode.*
3. *45 core clocks + 16 Northbridge clocks.*
4. *The latency of RDMSR and WRMSR are dependent on the particular machine register being accessed.*
5. *MONITOR and MWAIT are only used in multi-threaded environments; thus, the latency of the MONITOR/MWAIT idiom is highly variable. The latencies provided here are estimations of a lower bound. For MWAIT this encompasses the entire latency of the instruction (both the time before and after an incoming store to the monitored region).*
6. *The latency of this instruction is variable and depends on which register bits change.*

Table 14. System Instruction Latencies (Continued)

Syntax	Decode Type[1]	Latency	Notes
WRMSR	VectorPath		4

Note:

1. *For interpretation of special symbols, see "Decode Type" on page 247.*
2. *Values correspond to 64-bit mode/32-bit mode.*
3. *45 core clocks + 16 Northbridge clocks.*
4. *The latency of RDMSR and WRMSR are dependent on the particular machine register being accessed.*
5. *MONITOR and MWAIT are only used in multi-threaded environments; thus, the latency of the MONITOR/MWAIT idiom is highly variable. The latencies provided here are estimations of a lower bound. For MWAIT this encompasses the entire latency of the instruction (both the time before and after an incoming store to the monitored region).*
6. *The latency of this instruction is variable and depends on which register bits change.*

C.4 128-Bit Media Instruction Latencies

The table that follow provides the syntax, decode type, FPU pipes, latency, and throughput for the 128-bit media instructions, comprised of the SSE, SSE2, SSE3 and SSE4a instruction sets. For detailed information on the operation of these instructions, as well as opcodes, see the *AMD64 Architecture Programmer's Manual, Volume 4: 128-Bit Media Instructions*, order# 26568.

Table 15. 128-Bit Media Instruction Latencies

Syntax	Decode Type[7]	FPU Pipes	Lat	Through -put	Notes
ADDPD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD	4 (6)	1/1	
ADDPS *xmmreg1*, *xmmreg2 (mem*)	DirectPath Single★	FADD	4 (6)	1/1	
ADDSD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single	FADD	4 (6)	1/1	
ADDSS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single	FADD	4 (6)	1/1	
ADDSUBPD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD	4 (6)	1/1	
ADDSUBPS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD	4 (6)	1/1	
ANDNPD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
ANDNPS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
ANDPD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	

Notes:

1. *Also uses INT resources.*
2. *Uses multiple FP and INT resources.*
3. *Latency and throughput are only from FPU perspective and do not account for Load-Store unit and memory hierarchy complexities.*
4. *Does not use FPU execution units.*
5. *One 128-bit store requires two 64-bit access in the Load-Store Unit.*
6. *Latency and throughput assume aligned data and do not account for Load-Store Unit and memory hierarchy complexities. Unaligned locations typically add an extra cycle and halve the throughput.*
7. *For interpretation of special symbols, see "Decode Type" on page 247.*
8. *Uses multiple FP resources.*
9. *These instructions use the processor's write-combining resources.*

Table 15. 128-Bit Media Instruction Latencies (Continued)

Syntax	Decode Type[7]	FPU Pipes	Lat	Through -put	Notes
ANDPS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
CMPPD *xmmreg1*, *xmmreg2* (*mem*), *imm*	DirectPath Single★	FADD	2 (4)	1/1	
CMPPS *xmmreg1*, *xmmreg2* (*mem*), *imm*	DirectPath Single★	FADD	2 (4)	1/1	
CMPSD *xmmreg1*, *xmmreg2* (*mem*), *imm*	DirectPath Single	FADD	2 (4)	1/1	
CMPSS *xmmreg1*, *xmmreg2* (*mem*), *imm*	DirectPath Single	FADD	2 (4)	1/1	
COMISD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★★	FADD	3 (5)	1/1	
COMISS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★★	FADD	3 (5)	1/1	
CVTDQ2PD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FSTORE	4 (6)	1/1	
CVTDQ2PS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FSTORE	4 (6)	1/1	
CVTPD2DQ/ CVTTPD2DQ *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Double★	(FADD/FMUL) & FSTORE	7 (9)	1/1	
CVTPD2PI/ CVTTPD2PI *mmreg*, *xmmreg* (*mem*)	DirectPath Double★	(FADD/FMUL) & FSTORE	7 (9)	1/1	
CVTPD2PS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Double★	(FADD/FMUL) & FSTORE	7 (9)	1/1	

Notes:

1. *Also uses INT resources.*
2. *Uses multiple FP and INT resources.*
3. *Latency and throughput are only from FPU perspective and do not account for Load-Store unit and memory hierarchy complexities.*
4. *Does not use FPU execution units.*
5. *One 128-bit store requires two 64-bit access in the Load-Store Unit.*
6. *Latency and throughput assume aligned data and do not account for Load-Store Unit and memory hierarchy complexities. Unaligned locations typically add an extra cycle and halve the throughput.*
7. *For interpretation of special symbols, see “Decode Type” on page 247.*
8. *Uses multiple FP resources.*
9. *These instructions use the processor's write-combining resources.*

Table 15. 128-Bit Media Instruction Latencies (Continued)

Syntax	Decode Type[7]	FPU Pipes	Lat	Through -put	Notes
CVTPI2PD *xmmreg*, *mmreg* (*mem*)	DirectPath Single★	FSTORE	4 (6)	1/1	
CVTPI2PS *xmmreg, mmreg* (*mem*)	DirectPath Double▲	(FADD/FMUL) & FSTORE	7 (9)	1/1	
CVTPS2DQ/ CVTTPS2DQ *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FSTORE	4 (6)	1/1	
CVTPS2PD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FSTORE	2 (4)	1/1	
CVTPS2PI/ CVTTPS2PI *mmreg*, x*mmreg* (*mem*)	DirectPath Single	FSTORE	4 (6)	1/1	
CVTSD2SI/ CVTTSD2SI *reg*, *xmmreg* (*mem*)	DirectPath Double(★)	FADD & FSTORE	8 (10)	1/1	1
CVTSD2SS *xmmreg1, xmmreg2*	VectorPath	(FADD/FMUL) & FSTORE	8		
CVTSD2SS *xmmreg*, *mem*	DirectPath Double	(FADD/FMUL) & FSTORE	9	1/1	
CVTSI2SD *xmmreg*, *reg*	VectorPath ▲▲	(FADD/FMUL) & FSTORE	14		
CVTSI2SD *xmmreg*, *mem*	DirectPath Double▲	(FADD/FMUL) & FSTORE	9	1/1	
CVTSI2SS *xmmreg, reg*	VectorPath	(FADD/FMUL) & FSTORE	14		1

Notes:

1. *Also uses INT resources.*
2. *Uses multiple FP and INT resources.*
3. *Latency and throughput are only from FPU perspective and do not account for Load-Store unit and memory hierarchy complexities.*
4. *Does not use FPU execution units.*
5. *One 128-bit store requires two 64-bit access in the Load-Store Unit.*
6. *Latency and throughput assume aligned data and do not account for Load-Store Unit and memory hierarchy complexities. Unaligned locations typically add an extra cycle and halve the throughput.*
7. *For interpretation of special symbols, see “Decode Type” on page 247.*
8. *Uses multiple FP resources.*
9. *These instructions use the processor's write-combining resources.*

Table 15. 128-Bit Media Instruction Latencies (Continued)

Syntax	Decode Type[7]	FPU Pipes	Lat	Through -put	Notes
CVTSI2SS *xmmreg*, *mem*	DirectPath Double	(FADD/FMUL) & FSTORE	9	1/1	
CVTSS2SD *xmmreg1*, *xmmreg2*	VectorPath ▲▲	(FADD/FMUL) & FSTORE	7		
CVTSS2SD *xmmreg*, *mem*	DirectPath Double▲	(FADD/FMUL) & FSTORE	7	1/1	
CVTSS2SI/CVTTSS2SI *reg*, *xmmreg* (*mem*)	DirectPath Double	FADD & FSTORE	8 (10)	1/1	1
DIVPD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FMUL	20 (22)	1/17	
DIVPS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FMUL	18 (20)	1/15	
DIVSD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single	FMUL	20 (22)	1/17	
DIVSS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single	FMUL	16 (18)	1/13	
EXTRQ *xmmreg1*, *xmmreg2*	DirectPath Single	FADD/FMUL	2	2/1	
EXTRQ *xmmreg1*, *imm1*, *imm2*	DirectPath Single	FADD/FMUL	2	2/1	
FXRSTOR	VectorPath	–	89		
FXSAVE	VectorPath	–	63		
HADDPD *xmmreg1*, *xmmreg2*	DirectPath Single★	FADD	4	1/1	

Notes:

1. *Also uses INT resources.*
2. *Uses multiple FP and INT resources.*
3. *Latency and throughput are only from FPU perspective and do not account for Load-Store unit and memory hierarchy complexities.*
4. *Does not use FPU execution units.*
5. *One 128-bit store requires two 64-bit access in the Load-Store Unit.*
6. *Latency and throughput assume aligned data and do not account for Load-Store Unit and memory hierarchy complexities. Unaligned locations typically add an extra cycle and halve the throughput.*
7. *For interpretation of special symbols, see “Decode Type” on page 247.*
8. *Uses multiple FP resources.*
9. *These instructions use the processor's write-combining resources.*

Table 15. 128-Bit Media Instruction Latencies (Continued)

Syntax	Decode Type[7]	FPU Pipes	Lat	Through -put	Notes
HADDPD *xmmreg1*, *mem*	DirectPath Single★★	FADD	6	1/1	
HADDPS *xmmreg1*, *xmmreg2*	DirectPath Single★	FADD	4	1/1	
HADDPS *xmmreg1*, *mem*	DirectPath Single★★	FADD	6	1/1	
HSUBPD *xmmreg1*, *xmmreg2*	DirectPath Single★	FADD	4	1/1	
HSUBPD *xmmreg1*, *mem*	DirectPath Single★★	FADD	6	1/1	
HSUBPS *xmmreg1*, *xmmreg2*	DirectPath Single★	FADD	4	1/1	
HSUBPS *xmmreg1*, *mem*	DirectPath Single★★	FADD	6	1/1	
INSERTQ *xmmreg1*, *xmmreg2*	VectorPath	–	5		8
INSERTQ *xmmreg1*, *xmmreg2*, *imm1*, *imm2*	VectorPath	–	5		8
LDDQU *xmmreg*, *mem*	DirectPath Single★★		2	2/1	4, 6
LDMXCSR *mem*	VectorPath	–	12		2
MASKMOVDQU *xmmreg1*, *xmmreg2*	VectorPath	–			9
MAXPD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD	2 (4)	1/1	
MAXPS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD	2 (4)	1/1	

Notes:

1. *Also uses INT resources.*
2. *Uses multiple FP and INT resources.*
3. *Latency and throughput are only from FPU perspective and do not account for Load-Store unit and memory hierarchy complexities.*
4. *Does not use FPU execution units.*
5. *One 128-bit store requires two 64-bit access in the Load-Store Unit.*
6. *Latency and throughput assume aligned data and do not account for Load-Store Unit and memory hierarchy complexities. Unaligned locations typically add an extra cycle and halve the throughput.*
7. *For interpretation of special symbols, see “Decode Type” on page 247.*
8. *Uses multiple FP resources.*
9. *These instructions use the processor's write-combining resources.*

Table 15. 128-Bit Media Instruction Latencies (Continued)

Syntax	Decode Type[7]	FPU Pipes	Lat	Through -put	Notes
MAXSD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single	FADD	2 (4)	1/1	
MAXSS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single	FADD	2 (4)	1/1	
MINPD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD	2 (4)	1/1	
MINPS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD	2 (4)	1/1	
MINSD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single	FADD	2 (4)	1/1	
MINSS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single	FADD	2 (4)	1/1	
MOVAPD *xmmreg1*, *xmmreg2*	DirectPath Single★	FADD/FMUL/ FSTORE	2	3/1	
MOVAPD *mem*, *xmmreg*	DirectPath Double	FSTORE	2	1/1	3, 5
MOVAPD *xmmreg*, *mem*	DirectPath Single★		2	2/1	4
MOVAPS *xmmreg1*, *xmmreg2*	DirectPath Single★	FADD/FMUL/ FSTORE	2	3/1	
MOVAPS *mem*, *xmmreg*	DirectPath Double	FSTORE	2	1/1	3, 5
MOVAPS *xmmreg*, *mem*	DirectPath Single★		2	2/1	4
MOVD *xmmreg*, *reg*	DirectPath Double★		6		1, 4

Notes:

1. *Also uses INT resources.*
2. *Uses multiple FP and INT resources.*
3. *Latency and throughput are only from FPU perspective and do not account for Load-Store unit and memory hierarchy complexities.*
4. *Does not use FPU execution units.*
5. *One 128-bit store requires two 64-bit access in the Load-Store Unit.*
6. *Latency and throughput assume aligned data and do not account for Load-Store Unit and memory hierarchy complexities. Unaligned locations typically add an extra cycle and halve the throughput.*
7. *For interpretation of special symbols, see “Decode Type” on page 247.*
8. *Uses multiple FP resources.*
9. *These instructions use the processor's write-combining resources.*

Table 15. 128-Bit Media Instruction Latencies (Continued)

Syntax	Decode Type[7]	FPU Pipes	Lat	Through -put	Notes
MOVD *reg*, *xmmreg*	DirectPath Single★	FADD	3	1/1	1
MOVD *mem*, *xmmreg*	DirectPath Single	FSTORE	2	1/1	3
MOVD *xmmreg*, *mem*	DirectPath Single★		2	2/1	4
MOVDDUP *xmmreg1*, *xmmreg2*	DirectPath Single★	FADD/FMUL/ FSTORE	2	3/1	
MOVDDUP *xmmreg*, *mem*	DirectPath Single★		2	2/1	4
MOVDQ2Q *mmreg*, *xmmreg*	DirectPath Single	FADD/FMUL/ FSTORE	2	3/1	
MOVDQA *xmmreg1*, *xmmreg2*	DirectPath Single★	FADD/FMUL/ FSTORE	2	3/1	
MOVDQA *mem*, *xmmreg*	DirectPath Double	FSTORE	2	1/1	3, 5
MOVDQA *xmmreg*, *mem*	DirectPath Single★		2	2/1	4
MOVDQU *xmmreg1*, *xmmreg2*	DirectPath Single★	FADD/FMUL/ FSTORE	2	3/1	
MOVDQU *mem*, *xmmreg*	VectorPath	FSTORE	3	1/2	3
MOVDQU *xmmreg*, *mem*	DirectPath Single★★		2	2/1	4, 6
MOVHLPS *xmmreg1*, *xmmreg2*	DirectPath Single	FADD/FMUL	2	2/1	

Notes:

1. *Also uses INT resources.*
2. *Uses multiple FP and INT resources.*
3. *Latency and throughput are only from FPU perspective and do not account for Load-Store unit and memory hierarchy complexities.*
4. *Does not use FPU execution units.*
5. *One 128-bit store requires two 64-bit access in the Load-Store Unit.*
6. *Latency and throughput assume aligned data and do not account for Load-Store Unit and memory hierarchy complexities. Unaligned locations typically add an extra cycle and halve the throughput.*
7. *For interpretation of special symbols, see “Decode Type” on page 247.*
8. *Uses multiple FP resources.*
9. *These instructions use the processor's write-combining resources.*

Table 15. 128-Bit Media Instruction Latencies (Continued)

Syntax	Decode Type[7]	FPU Pipes	Lat	Through -put	Notes
MOVHPD *mem*, *xmmreg*	DirectPath Single	FSTORE	2	1/1	3
MOVHPD *xmmreg*, *mem*	DirectPath Single	FADD/FMUL	4	2/1	
MOVHPS *mem, xmmreg*	DirectPath Single	FSTORE	2	1/1	3
MOVHPS *xmmreg*, *mem*	DirectPath Single	FADD/FMUL	4	2/1	
MOVLHPS *xmmreg1*, *xmmreg2*	DirectPath Single	FADD/FMUL	2	2/1	
MOVLPD *mem*, *xmmreg*	DirectPath Single	FSTORE	2	1/1	3
MOVLPD *xmmreg*, *mem*	DirectPath Single	FADD/FMUL	4	2/1	
MOVLPS *mem, xmmreg*	DirectPath Single	FSTORE	2	1/1	3
MOVLPS *xmmreg*, *mem*	DirectPath Single	FADD/FMUL	4	2/1	
MOVMSKPD *reg*, *xmmreg*	DirectPath Single★★	FADD	3	1/1	
MOVMSKPS *reg*, *xmmreg*	DirectPath Single★★	FADD	3	1/1	
MOVNTDQ *mem*, *xmmreg*	DirectPath Double	FSTORE			5, 9
MOVNTPD *mem*, *xmmreg*	DirectPath Double	FSTORE			5, 9

Notes:

1. *Also uses INT resources.*
2. *Uses multiple FP and INT resources.*
3. *Latency and throughput are only from FPU perspective and do not account for Load-Store unit and memory hierarchy complexities.*
4. *Does not use FPU execution units.*
5. *One 128-bit store requires two 64-bit access in the Load-Store Unit.*
6. *Latency and throughput assume aligned data and do not account for Load-Store Unit and memory hierarchy complexities. Unaligned locations typically add an extra cycle and halve the throughput.*
7. *For interpretation of special symbols, see “Decode Type” on page 247.*
8. *Uses multiple FP resources.*
9. *These instructions use the processor's write-combining resources.*

Table 15. 128-Bit Media Instruction Latencies (Continued)

Syntax	Decode Type[7]	FPU Pipes	Lat	Through -put	Notes
MOVNTPS *mem*, *xmmreg*	DirectPath Double	FSTORE			5, 9
MOVNTSD *mem*, *xmmreg*	DirectPath Single	FSTORE			9
MOVNTSS *mem*, *xmmreg*	DirectPath Single	FSTORE			9
MOVQ *xmmreg1*, *xmmreg2*	DirectPath Single★	FADD/FMUL/ FSTORE	2	3/1	
MOVQ *mem*, *xmmreg*	DirectPath Single	FSTORE	2	1/1	3
MOVQ *xmmreg*, *mem*	DirectPath Single★		2	2/1	4
MOVQ2DQ *xmmreg, mmreg*	DirectPath Single★	FADD/FMUL/ FSTORE	2	3/1	
MOVSD *xmmreg1*, *xmmreg2*	DirectPath Single	FADD/FMUL	2	2/1	
MOVSD *mem*, *xmmreg*	DirectPath Single	FSTORE	2	1/1	3
MOVSD *xmmreg*, *mem*	DirectPath Single★		2	2/1	4
MOVSHDUP *xmmreg1*, *xmmreg2*	DirectPath Single★	FADD/FMUL	2	2/1	
MOVSHDUP *xmmreg*, *mem*	DirectPath Single★		2	2/1	4
MOVSLDUP *xmmreg1*, *xmmreg2*	DirectPath Single★	FADD/FMUL	2	2/1	

Notes:

1. *Also uses INT resources.*
2. *Uses multiple FP and INT resources.*
3. *Latency and throughput are only from FPU perspective and do not account for Load-Store unit and memory hierarchy complexities.*
4. *Does not use FPU execution units.*
5. *One 128-bit store requires two 64-bit access in the Load-Store Unit.*
6. *Latency and throughput assume aligned data and do not account for Load-Store Unit and memory hierarchy complexities. Unaligned locations typically add an extra cycle and halve the throughput.*
7. *For interpretation of special symbols, see “Decode Type” on page 247.*
8. *Uses multiple FP resources.*
9. *These instructions use the processor's write-combining resources.*

Table 15. 128-Bit Media Instruction Latencies (Continued)

Syntax	Decode Type[7]	FPU Pipes	Lat	Through -put	Notes
MOVSLDUP *xmmreg*, *mem*	DirectPath Single★		2	2/1	4
MOVSS *xmmreg1*, *xmmreg2*	DirectPath Single	FADD/FMUL	2	2/1	
MOVSS *mem*, *xmmreg*	DirectPath Single	FSTORE	2	1/1	3
MOVSS *xmmreg*, *mem*	DirectPath Single★		2	2/1	4
MOVUPD *xmmreg1*, *xmmreg2*	DirectPath Single★	FADD/FMUL/ FSTORE	2	3/1	
MOVUPD *mem*, *xmmreg*	VectorPath	FSTORE	3	1/2	3
MOVUPD *xmmreg*, *mem*	DirectPath Single★★		2	2/1	4, 6
MOVUPS *xmmreg1*, *xmmreg2*	DirectPath Single★	FADD/FMUL/ FSTORE	2	3/1	
MOVUPS *mem*, *xmmreg*	VectorPath	FSTORE	3	1/2	3
MOVUPS *xmmreg*, *mem*	DirectPath Single★★		2	2/1	4, 6
MULPD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FMUL	4 (6)	1/1	
MULPS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single ★	FMUL	4 (6)	1/1	
MULSD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single	FMUL	4 (6)	1/1	
MULSS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single	FMUL	4 (6)	1/1	

Notes:

1. *Also uses INT resources.*
2. *Uses multiple FP and INT resources.*
3. *Latency and throughput are only from FPU perspective and do not account for Load-Store unit and memory hierarchy complexities.*
4. *Does not use FPU execution units.*
5. *One 128-bit store requires two 64-bit access in the Load-Store Unit.*
6. *Latency and throughput assume aligned data and do not account for Load-Store Unit and memory hierarchy complexities. Unaligned locations typically add an extra cycle and halve the throughput.*
7. *For interpretation of special symbols, see “Decode Type” on page 247.*
8. *Uses multiple FP resources.*
9. *These instructions use the processor's write-combining resources.*

Table 15. 128-Bit Media Instruction Latencies (Continued)

Syntax	Decode Type[7]	FPU Pipes	Lat	Through -put	Notes
ORPD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
ORPS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PACKSSDW/PACKSSWB/ PACKUSWB *xmmreg1*, *xmmreg1* (*mem*)	DirectPath Single★★	FADD/FMUL	2 (4)	2/1	
PADDB/PADDW/PADDD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PADDQ *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PADDSB/PADDSW *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PADDUSB/ PADDUSW *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PAND *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PANDN *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PAVGB/ PAVGW *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PCMPEQB/PCMPEQW/ PCMPEQD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PCMPGTB/PCMPGTW/ PCMPGTD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PEXTRW *reg*, *xmmreg*, *imm*	DirectPath Double	FADD & FSTORE	6	1/1	1

Notes:

1. *Also uses INT resources.*
2. *Uses multiple FP and INT resources.*
3. *Latency and throughput are only from FPU perspective and do not account for Load-Store unit and memory hierarchy complexities.*
4. *Does not use FPU execution units.*
5. *One 128-bit store requires two 64-bit access in the Load-Store Unit.*
6. *Latency and throughput assume aligned data and do not account for Load-Store Unit and memory hierarchy complexities. Unaligned locations typically add an extra cycle and halve the throughput.*
7. *For interpretation of special symbols, see “Decode Type” on page 247.*
8. *Uses multiple FP resources.*
9. *These instructions use the processor's write-combining resources.*

Table 15. 128-Bit Media Instruction Latencies (Continued)

Syntax	Decode Type[7]	FPU Pipes	Lat	Through -put	Notes
PINSRW *xmmreg*, *reg*, *imm*	DirectPath Double★	FADD/FMUL	9		1
PINSRW *xmmreg*, *mem*, *imm*	DirectPath Single★	FADD/FMUL	4	1/1	
PMADDWD *xmmreg1*, *xmmreg2 (mem*)	DirectPath Single★	FMUL	3 (5)	1/1	
PMAXSW/ PMAXUB *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PMINSW/ PMINUB *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PMOVMSKB *reg*, *xmmreg*	DirectPath Single★★	FADD	3	1/1	
PMULHUW/ PMULHW *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FMUL	3 (5)	1/1	
PMULLW *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FMUL	3 (5)	1/1	
PMULUDQ *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FMUL	3 (5)	1/1	
POR *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PSADBW *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD	3 (5)	1/1	
PSHUFD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★★	FADD/FMUL	2 (4)	2/1	
PSHUFHW *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	

Notes:

1. *Also uses INT resources.*
2. *Uses multiple FP and INT resources.*
3. *Latency and throughput are only from FPU perspective and do not account for Load-Store unit and memory hierarchy complexities.*
4. *Does not use FPU execution units.*
5. *One 128-bit store requires two 64-bit access in the Load-Store Unit.*
6. *Latency and throughput assume aligned data and do not account for Load-Store Unit and memory hierarchy complexities. Unaligned locations typically add an extra cycle and halve the throughput.*
7. *For interpretation of special symbols, see “Decode Type” on page 247.*
8. *Uses multiple FP resources.*
9. *These instructions use the processor's write-combining resources.*

Table 15. 128-Bit Media Instruction Latencies (Continued)

Syntax	Decode Type[7]	FPU Pipes	Lat	Through -put	Notes
PSHUFLW *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PSLLW/PSLLD/ PSLLQ *xmmreg1*, *xmmreg2/imm* (mem)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PSLLDQ *xmmreg*, *imm*	DirectPath Single★	FADD/FMUL	2	2/1	
PSRAW/ PSRAD *xmmreg1*, *xmmreg2/imm* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PSRLW/PSRLD/ PSRLQ *xmmreg1*, *xmmreg2/imm* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PSRLDQ *xmmreg*, *imm*	DirectPath Single★	FADD/FMUL	2	2/1	
PSUBB/PSUBW/PSUBD/ PSUBQ *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PSUBSB/ PSUBSW *xmmreg1*, *xmmreg2* (*mem)*	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PSUBUSB/ PSUBUSW *xmmreg1*, *xmmreg2* (*mem)*	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/ PUNPCKHQDQ *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PUNPCKLBW/PUNPCKLWD/ PUNPCKLDQ *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
PUNPCKLQDQ *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PXOR *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	

Notes:

1. *Also uses INT resources.*
2. *Uses multiple FP and INT resources.*
3. *Latency and throughput are only from FPU perspective and do not account for Load-Store unit and memory hierarchy complexities.*
4. *Does not use FPU execution units.*
5. *One 128-bit store requires two 64-bit access in the Load-Store Unit.*
6. *Latency and throughput assume aligned data and do not account for Load-Store Unit and memory hierarchy complexities. Unaligned locations typically add an extra cycle and halve the throughput.*
7. *For interpretation of special symbols, see “Decode Type” on page 247.*
8. *Uses multiple FP resources.*
9. *These instructions use the processor's write-combining resources.*

Table 15. 128-Bit Media Instruction Latencies (Continued)

Syntax	Decode Type[7]	FPU Pipes	Lat	Through -put	Notes
RCPPS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FMUL	3 (5)	1/1	
RCPSS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single	FMUL	3 (5)	1/1	
RSQRTPS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FMUL	3 (5)	1/1	
RSQRTSS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single	FMUL	3 (5)	1/1	
SHUFPD *xmmreg1*, *xmmreg2* (*mem*), imm8	DirectPath Single★★	FADD/FMUL	2 (4)	2/1	
SHUFPS *xmmreg1*, *xmmreg2* (*mem*), imm8	DirectPath Single★★	FADD/FMUL	2 (4)	2/1	
SQRTPD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FMUL	27 (29)	1/24	
SQRTPS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FMUL	21 (23)	1/18	
SQRTSD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single	FMUL	27 (29)	1/24	
SQRTSS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single	FMUL	19 (21)	1/16	
STMXCSR	VectorPath		12		
SUBPD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD	4 (6)	1/1	
SUBPS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD	4 (6)	1/1	

Notes:

1. *Also uses INT resources.*
2. *Uses multiple FP and INT resources.*
3. *Latency and throughput are only from FPU perspective and do not account for Load-Store unit and memory hierarchy complexities.*
4. *Does not use FPU execution units.*
5. *One 128-bit store requires two 64-bit access in the Load-Store Unit.*
6. *Latency and throughput assume aligned data and do not account for Load-Store Unit and memory hierarchy complexities. Unaligned locations typically add an extra cycle and halve the throughput.*
7. *For interpretation of special symbols, see “Decode Type” on page 247.*
8. *Uses multiple FP resources.*
9. *These instructions use the processor's write-combining resources.*

Table 15. 128-Bit Media Instruction Latencies (Continued)

Syntax	Decode Type[7]	FPU Pipes	Lat	Through -put	Notes
SUBSD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single	FADD	4 (6)	1/1	
SUBSS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single	FADD	4 (6)	1/1	
UCOMISD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★★	FADD	3 (5)	1/1	
UCOMISS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★★	FADD	3 (5)	1/1	
UNPCKHPD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
UNPCKHPS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
UNPCKLPD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
UNPCKLPS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
XORPD *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	
XORPS *xmmreg1*, *xmmreg2* (*mem*)	DirectPath Single★	FADD/FMUL	2 (4)	2/1	

Notes:

1. *Also uses INT resources.*
2. *Uses multiple FP and INT resources.*
3. *Latency and throughput are only from FPU perspective and do not account for Load-Store unit and memory hierarchy complexities.*
4. *Does not use FPU execution units.*
5. *One 128-bit store requires two 64-bit access in the Load-Store Unit.*
6. *Latency and throughput assume aligned data and do not account for Load-Store Unit and memory hierarchy complexities. Unaligned locations typically add an extra cycle and halve the throughput.*
7. *For interpretation of special symbols, see “Decode Type” on page 247.*
8. *Uses multiple FP resources.*
9. *These instructions use the processor's write-combining resources.*

C.5 64-Bit Media Instruction Latencies

The 64-bit media instructions consist of the AMD 3DNow!™ instructions and AMD 3DNow! extensions and the MMX™ instructions and MMX extensions. The following tables provide the decode type, FPU pipe(s), latency and throughput corresponding to each instruction mnemonic. For more detailed information on the operation of a particular instruction, as well as encoding

information, see the *AMD64 Architecture Programmer's Manual Volume 5: 64-Bit Media and x87 Instructions*, order# 26569.

Table 16. 64-Bit Media Instruction Latencies

Syntax	Decode Type[2]	FPU Pipe(s)	Lat	Through -put	Notes
CVTPD2PI/ CVTTPD2PI *mmreg*, *xmmreg* (*mem*)	DirectPath Double★	(FADD/FMUL) & FSTORE	7 (9)	1/1	
CVTPI2PD *xmmreg*, *mmreg* (*mem*)	DirectPath Single★	FSTORE	4 (6)	1/1	
CVTPI2PS *xmmreg, mmreg* (*mem*)	DirectPath Double▲	(FADD/FMUL) & FSTORE	7 (9)	1/1	
CVTPS2PI/ CVTTPS2PI *mmreg*, x*mmreg* (*mem*)	DirectPath Single	FSTORE	4 (6)	1/1	
EMMS	DirectPath Single	FADD/FMUL/ FSTORE	2		
FEMMS	DirectPath Single	FADD/FMUL/ FSTORE	2		
FRSTOR	VectorPath	–	133		
FSAVE (FNSAVE)	VectorPath	–	162		
FXRSTOR	VectorPath	–	89		
FXSAVE	VectorPath	–	63		
MASKMOVQ *mmreg1*, *mmreg2*	VectorPath				3
MOVD *mmreg*, *reg*	DirectPath Double	–	6		
MOVD *reg*, *mmreg*	DirectPath Single★	–	3		
MOVD *mmreg*, *mem*	DirectPath Single	FADD/FMUL/ FSTORE	4		
MOVD *mem*, *mmreg*	DirectPath Single	FSTORE	2		

Notes:

1. *Also uses INT resources.*
2. *For interpretation of special symbols, see "Decode Type" on page 247.*
3. *Uses multiple INT and FP resources.*
4. *These instructions use the processor's write-combining resources.*

Table 16. 64-Bit Media Instruction Latencies (Continued)

Syntax	Decode Type[2]	FPU Pipe(s)	Lat	Through -put	Notes
MOVDQ2Q *mmreg*, *xmmreg*	DirectPath Single	FADD/FMUL/ FSTORE	2	3/1	
MOVNTQ *mem*, *mmreg*	DirectPath Single	FSTORE			4
MOVQ *mmreg1*, *mmreg2*	DirectPath Single	FADD/FMUL	2		
MOVQ *mmreg*, *mem*	DirectPath Single	FADD/FMUL/ FSTORE	4		
MOVQ *mem*, *mmreg*	DirectPath Single	FSTORE	2		
MOVQ2DQ *xmmreg, mmreg*	DirectPath Single★	FADD/FMUL/ FSTORE	2	3/1	
PACKSSDW/PACKSSWB/ PACKUSWB *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PADDB/PADDW/PADDD/PADDQ *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PADDSB/PADDSW *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PADDUSB/PADDUSW *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PAND *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PANDN *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PAVGB/ PAVGW/PAVGUSB *mmreg1*, *mmreg2* (*mem)*	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PCMPEQB/PCMPEQW/ PCMPEQD *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PCMPGTB/PCMPGTW/ PCMPGTD *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PEXTRW *reg*, *mmreg*, *imm*	DirectPath Double	FADD & FSTORE	6	1/1	1

Notes:

1. *Also uses INT resources.*
2. *For interpretation of special symbols, see “Decode Type” on page 247.*
3. *Uses multiple INT and FP resources.*
4. *These instructions use the processor's write-combining resources.*

Table 16. 64-Bit Media Instruction Latencies (Continued)

Syntax	Decode Type[2]	FPU Pipe(s)	Lat	Through -put	Notes
PF2ID *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD	4 (6)	1/1	
PF2IW *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD	4 (6)	1/1	
PFACC *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD	4 (6)	1/1	
PFADD *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD	4 (6)	1/1	
PFCMPEQ *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD	2 (4)	1/1	
PFCMPGE *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD	2 (4)	1/1	
PFCMPGT *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD	2 (4)	1/1	
PFMAX *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD	2 (4)	1/1	
PFMIN *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD	2 (4)	1/1	
PFMUL *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FMUL	4 (6)	1/1	
PFNACC *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD	4 (6)	1/1	
PFPNACC *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD	4 (6)	1/1	
PFRCP *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FMUL	3 (5)	1/1	
PFRCPIT1 *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FMUL	4 (6)	1/1	
PFRCPIT2 *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FMUL	4 (6)	1/1	
PFRSQIT1 *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FMUL	4 (6)	1/1	

Notes:

1. *Also uses INT resources.*
2. *For interpretation of special symbols, see “Decode Type” on page 247.*
3. *Uses multiple INT and FP resources.*
4. *These instructions use the processor's write-combining resources.*

Table 16. 64-Bit Media Instruction Latencies (Continued)

Syntax	Decode Type[2]	FPU Pipe(s)	Lat	Through -put	Notes
PFRSQRT *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FMUL	3 (5)	1/1	
PFSUB *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD	4 (6)	1/1	
PFSUBR *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD	4 (6)	1/1	
PI2FD *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD	4 (6)	1/1	
PI2FW *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD	4 (6)	1/1	
PINSRW *mmreg*, *reg*, *imm*	DirectPath Double	FADD/FMUL	9		1
PINSRW *mmreg*, *mem*, *imm*	DirectPath Single	FADD/FMUL	4	2/1	
PMADDWD *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FMUL	3 (5)	1/1	
PMAXSW/ PMAXUB *mmreg1*, *mmreg2* (*mem*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PMINSW/PMINUB *mmreg1*, *mmreg2* (*mem*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PMOVMSKB *reg*, *mmreg*	DirectPath Single★★	FADD	3	1/1	
PMULHRW *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FMUL	3 (5)	1/1	
PMULHUW *mmreg1*, *mmreg2* (*mem*)	DirectPath Single	FMUL	3 (5)	1/1	
PMULHW *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FMUL	3 (5)	1/1	
PMULLW *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FMUL	3 (5)	1/1	
PMULUDQ *mmreg1*, *mmreg2* (*mem*)	DirectPath Single	FMUL	3 (5)	1/1	

Notes:

1. *Also uses INT resources.*
2. *For interpretation of special symbols, see “Decode Type” on page 247.*
3. *Uses multiple INT and FP resources.*
4. *These instructions use the processor's write-combining resources.*

Table 16. 64-Bit Media Instruction Latencies (Continued)

Syntax	Decode Type[2]	FPU Pipe(s)	Lat	Through -put	Notes
POR *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PSADBW *mmreg1*, *mmreg2* (*mem*)	DirectPath Single	FADD	3 (5)	1/1	
PSHUFW *mmreg1*, *mmreg2* (*mem*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PSLLW/PSLLD/PSLLQ *mmreg1*, *mmreg2/imm* (*mem64*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PSRAW/PSRAD *mmreg1*, *mmreg2/imm* (*mem64*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PSRLW/PSRLD/PSRLQ *mmreg1*, *mmreg2/imm* (*mem64*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PSUBB/PSUBW/PSUBD/ PSUBQ *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PSUBSB/ PSUBSW*mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PSUBUSB/ PSUBUSW *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PSWAPD *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD/MUL	2 (4)	2/1	
PUNPCKHBW/PUNPCKHWD/ PUNPCKHDQ *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PUNPCKLBW/PUNPCKLWD/ PUNPCKLDQ *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	
PXOR *mmreg1*, *mmreg2* (*mem64*)	DirectPath Single	FADD/FMUL	2 (4)	2/1	

Notes:

1. *Also uses INT resources.*
2. *For interpretation of special symbols, see “Decode Type” on page 247.*
3. *Uses multiple INT and FP resources.*
4. *These instructions use the processor's write-combining resources.*

C.6 x87 Floating-Point Instruction Latencies

The following tables provide the decode type, FPU pipe(s), latency and throughput corresponding to each x87 floating-point instruction mnemonic. For more detailed information on the operation of a particular instruction, as well as encoding information, see the *AMD64 Architecture Programmer’s Manual Volume 5: 64-Bit Media and x87 Instructions*, order# 26569.

Table 17. x87 Floating-Point Instruction Latencies

Instruction	Decode Type	FPU Pipes	Latency
F2XM1	VectorPath	–	65
FABS	DirectPath Single	FMUL	2
FADD/FADDP ST(*i*)	DirectPath Single	FADD	4
FADD/FADDP *mem32/64*	DirectPath Single	FADD	6
FBLD	VectorPath	–	94
FBSTP	VectorPath	–	160
FCHS	DirectPath Single	FMUL	2
FCMOV*cc* ST(*i*)	VectorPath	–	15
FCOM/FCOMP/FCOMPP	DirectPath Single	FADD	2
FCOM/FCOMP ST(*i*)	DirectPath Single	FADD	2
FCOM/FCOMP *mem32/64*	DirectPath Single	FADD	4
FCOMI/FCOMIP ST(*i*)	VectorPath	FADD	3
FCOS	VectorPath	–	93
FDECSTP	DirectPath Single	FADD/FMUL/FSTORE	2
FDIV/FDIVP/FDIVR/FDIVRP ST(*i*)	DirectPath Single	FMUL	16/20/24
FDIV/FDIVR *mem32/64*	DirectPath Single	FMUL	18/22/26
FFREE ST(*i*)	DirectPath Single	FADD/FMUL/FSTORE	2
FIADD *mem16/32*	DirectPath Double	–	11
FICOM/FICOMP *mem16/32*	DirectPath Double	–	9
FIDIV/FIDIVR *mem16/32*	DirectPath Double	–	31
FILD *mem16/32/64*	DirectPath Single	FSTORE	6
FIMUL *mem16/32*	DirectPath Double	–	11
FINCSTP	DirectPath Single	FADD/FMUL/FSTORE	2
FIST/FISTP *mem16/32/64*	DirectPath Single	FSTORE	4
FISTTP *mem*	DirectPath Single	FSTORE	4
FISUB/FISUBR *mem16/32*	DirectPath Double	–	11

Notes:

1. *There is additional latency associated with this instruction. "e" represents the difference between the exponents of the divisor and the dividend. If "s" is the number of normalization shifts performed on the result, then $n = (s+1)/2$ where $(0 \leq n \leq 32)$.*

Table 17. x87 Floating-Point Instruction Latencies (Continued)

Instruction	Decode Type	FPU Pipes	Latency
FLD ST(*i*)	DirectPath Single	FADD/FMUL	2
FLD *mem32/64*	DirectPath Single	FADD/FMUL/FSTORE	4
FLD *mem80*	VectorPath	FADD/FMUL	13
FLD1/FLDL2E/FLDL2T/FLDLG2/ FLDLN2/FLDPI/FLDZ	DirectPath Single	FSTORE	4
FLDCW	VectorPath	–	12
FLDENV	VectorPath	–	116
FMUL/FMULP ST(*i*)	DirectPath Single	FMUL	4
FMUL/FMULP *mem32/64*	DirectPath Single	FMUL	6
FNCLEX	VectorPath	–	17
FNINIT	VectorPath	–	92
FNOP	DirectPath Single	FADD/FMUL/FSTORE	2
FNSAVE	VectorPath	–	162
FNSTCW	VectorPath	–	2
FNSTENV	VectorPath	–	76
FNSTSW AX	VectorPath	–	9
FNSTSW *mem*	VectorPath	–	4
FPATAN	VectorPath	–	151
FPREM	DirectPath Single	FMUL	$9+e+n^{1}$
FPREM1	DirectPath Single	FMUL	$9+e+n^{1}$
FPTAN	VectorPath	–	109
FRNDINT	VectorPath	–	10
FRSTOR	VectorPath	–	132
FSCALE	VectorPath	–	9
FSIN	VectorPath	–	93
FSINCOS	VectorPath	–	105
FSQRT ST(*i*)	DirectPath Single	FMUL	19/27/35

Notes:

1. *There is additional latency associated with this instruction. "e" represents the difference between the exponents of the divisor and the dividend. If "s" is the number of normalization shifts performed on the result, then n = (s+1)/2 where (0 ≤ n ≤ 32).*

Table 17. x87 Floating-Point Instruction Latencies (Continued)

Instruction	Decode Type	FPU Pipes	Latency
FST/FSTP ST(*i*)	DirectPath Single	FADD/FMUL	2
FST/FSTP *mem32/64*	DirectPath Single	FSTORE	2
FSTP *mem80*	VectorPath	–	8
FSUB/FSUBP/FSUBR/FSUBRP ST(*i*)	DirectPath Single	FADD	4
FSUB/FSUBR *mem32/64*	DirectPath Single	FADD	6
FTST	DirectPath Single	FADD	2
FUCOM/FUCOMP/FUCOMPP	DirectPath Single	FADD	2
FUCOMI/FUCOMIP ST(*i*)	VectorPath	FADD	3
FWAIT	DirectPath Single	–	~0
FXAM	VectorPath	–	2
FXCH ST(*i*)	DirectPath Single	FADD/FMUL/FSTORE	2
FXRSTOR	VectorPath	–	87
FXSAVE	VectorPath	–	60
FXTRACT	VectorPath	–	9
FYL2X	VectorPath	–	13
FYL2XP1	VectorPath	–	114

Notes:

1. *There is additional latency associated with this instruction. "e" represents the difference between the exponents of the divisor and the dividend. If "s" is the number of normalization shifts performed on the result, then n = (s+1)/2 where (0 ≤ n ≤ 32).*

Appendix D Tools and APIs for AMD Family 10h and 12h ccNUMA Multiprocessor Systems

The following sections discuss tools and APIs available to support AMD Family 10h and 12h ccNUMA multiprocessor systems.

D.1 Thread/Process Scheduling, Memory Affinity

This following sections discuss tools and APIs available for assigning thread/process and memory affinity under various operating systems.

D.1.1 Support Under Linux®

Linux provides command-line utilities to explicitly set process/thread and memory affinity to both nodes and cores on a node. Additionally, **libnuma**, a shared library, is provided for more precise affinity control from within applications.

D.1.1.1 Controlling Process and Thread Affinity

The Linux command-line utlities offer high-level affinity control options. The **numactl** utility is a command line tool for running a process with a specific *node* affinity.

For example, to run the `foobar` program on the cores of node 0, enter the following at the command prompt:

```
numactl --cpubind=0  foobar
```

Application and kernel developers can use the **libnuma** shared library, which can be linked to programs and offers a stable API for setting thread affinity to a given node or set of nodes. Interested developers should consult the Linux **man** pages for details on the various functions available.

On a quad-core processor, a process or thread affined to a particular node using the tools or API discussed above may still migrate back and forth between the cores of that node. This migration may or may not affect performance.

The **taskset** utility is a command-line tool for setting the process affinity for a specified program to any core. For example, to run the `foobar` program on the first two cores of node 0, enter the following on the command line:

```
taskset -c 0,1  foobar
```

In SuSE Linux Enterprise Server 10/10.1, the **numactl** utility can be used instead of **taskset** to set process affinity to any core.

Linux provides several functions by which to set the thread affinity to any core or set of cores:

- **pthread_attr_setaffinity_np()** and **pthread_create()** are provided as a part of the older **nptl** library; they can be used to set the affinity parameter and then create a thread using that affinity.
- **sched_setaffinity()** system call and **schedutils** scheduler utilities.

D.1.1.2 Controlling Memory Affinity

Both **numactl** and **libnuma** library functions can be used to set memory affinity[5]. Memory affinity set by tools like **numactl** applies to all the data accessed by the entire program (including child processes). Memory affinity set by **libnuma** or other library functions can be made to apply only to specific data as determined by the program.

Both **numactl** and the **libnuma** API can be used to set a preferred memory affinity instead of forcibly binding it. In this case the binding specified is a hint to the OS; the OS may choose not to adhere to it.

At a high level, normal first touch binding, explicit binding and preferred binding are all available as memory policies on Linux.

By default, when none of the tools/API is used, Linux uses the first touch binding policy for all data. Once memory is bound, either by the OS, or by using the tools/API, the memory will normally remain resident on that node for its lifetime.

D.1.2 Support under Solaris™

Sun Solaris™ provides several tools and API's for influencing thread/process and memory affinity[6].

Solaris provides a command line tool called **pbind** to set process affinity. There is also a shared library called **liblgrp** that provides an API that a program can call to set thread affinity.

Solaris provides a memory placement API to affect memory placement. A program can call the **madvise()** function to provide hints to the OS as to the memory policy to use. This API does not allow binding of memory to an explicit node or set of nodes specified on the command line or in the program. But there are several policies other than the first touch policy that can be used.

For example, a thread can use **madvise** to migrate the data it needs to the node where it runs, instead of leaving it on a different node, on which it was first touched by another thread. There is, naturally, a cost associated with the migration.

Solaris provides a library called **madv.so.1** that can interpose on memory allocation system calls and call the madvise function internally for the memory policy.

By default, Solaris uses the first touch binding policy for data that is not shared. Once memory is bound to a node it normally remains resident on that node for its lifetime.

Sun is also working on supporting several command line tools to control thread and memory placement. These are expected to be integrated in the upcoming versions of Solaris, but experimental versions are currently available[7].

D.1.3 Support under Microsoft® Windows®

In the Microsoft Windows environment, the function to bind a thread on particular core or cores is **SetThreadAffinityMask()**. The function to run all threads in a process on particular core or cores is **SetProcessAffinityMask()**[8].

The function to set memory affinity for a thread is **VirtualAlloc()**[9]. This function gives the developer the choice to bind memory immediately on allocation or to defer binding until first touch.

The start /affinity xxx command can be used to confine all of a process's threads to a specified subset of cores in the system. The memory that these threads allocate or touch will also be confined to that subset of cores. In addition, several Microsoft Enterprise products provide NUMA support and configurability, such as SQL Server 2005 [10] and IIS [11].

If an application relies on heaps in Windows, we recommend using a low fragmentation heap (LFH) and using a local heap instead of a global heap[12][13].

By default, Windows uses the first touch binding policy for all data. Once memory is bound to a node, it normally resides on that node for its lifetime.

D.2 Tools and APIs for Node Interleaving

This section discusses tools and APIs available for performing node interleaving under various operating systems.

D.2.1 Support under Linux®

Linux provides several ways for an application to use node interleaving [5].

- **numactl** is a command line tool, which is used for node interleaving all memory accessed by a program across a set of chosen nodes.

For example, to interleave all memory accessed by program `foobar` on nodes 0 and 1, use:

```
numactl --interleave=0x03  foobar
```

- **libnuma** offers several functions a program can use to interleave a given memory region across a set of chosen nodes.

Linux only supports the round robin node interleaving policy.

D.2.2 Support under Solaris™

Solaris offers an API called **madvise**, which can be usedwith the **MADV_ACCESS_MANY** flag to tell the OS to use a memory policy that causes the OS to bind memory randomly across the nodes. This offers behavior similar to the round robin node interleaving of memory offered by Linux.

This random policy is the default memory placement policy used by Solaris for shared memory.

D.2.3 Support under Microsoft® Windows®

Microsoft Windows does not offer node interleaving.

D.2.4 Node Interleaving Configuration in the BIOS

AMD Family 10h and 12h ccNUMA multiprocessor systems can be configured in the BIOS to interleave all memory across all nodes on a page basis (4KB for regular pages and 2M for large pages). Enabling node interleaving in the BIOS overrides the use of any tools and causes the OS to interleave all available system memory across all nodes in a round robin manner.

Appendix E NUMA Optimizations for I/O Devices

E.1 AMD64 System Topologies

AMD Family 10h and 12h, and AMD Opteron™ systems range from single-node desktop to two, four, and eight node systems. Each node in the AMD Family 10h and 12h system consists of four CPU cores attached to an integrated memory controller and up to four HyperTransport™ links. I/O devices (PCIe™, AGP, PCI-X, and PCI) connect to the system over non-coherent HyperTransport™ links. The term *non-coherent I/O* refers to a configuration in which the CPU does not cache memory residing on an I/O device and in which the I/O device does not cache system memory shared by the CPU(s).

The integrated Northbridge converts requests issued on the non-coherent HyperTransport link into coherent requests before forwarding them into the CPU coherent fabric. Since memory is sharable and coherent to all of the CPUs in the system, it is possible that the latest copy of the requested memory location is not in memory, but is located in one of the CPU caches, if that memory location is cached and that cache-line has subsequently been modified by one of the CPU cores. In this case, the CPU core holding the data must recognize that it is the owner of the data and must return this data to the I/O device. This is often referred to as *probing* the caches. This implies that the latency to obtain the data will be directly affected by the location of the latest data at the time the I/O device requests it.

There are no existing API/BIOS interfaces that allow the association of a particular I/O device with a specific node in a NUMA system or that provide detailed information on device-to-node topology. Thus, developers should adhere to the guidelines provided in the following sections.

E.2 Key Optimizations

The OS can help manage device NUMA topology information to supplement existing OS NUMA support by means of affinity for device-driver buffers and resources (interrupt-pin assignment, ISR, DPC), when device drivers are loaded. The application I/O-thread is the portion of application code that calls the device's I/O API, which in turn will call the I/O device-driver. On AMD Family 10h and 12h and AMD Opteron systems, applications that interface with an I/O device (through API/device drivers) usually perform best when the following conditions are true:

- The application or device driver uses a hardware device on one of the I/O buses and the device depends upon system memory latency/bandwidth as a primary performance determinant.

 For example, a device that processes operations from its own adapter memory is not as sensitive to system memory latency as an adapter that primarily accesses system memory. Latencies of local memory reads by I/O devices can be ~10%–25% more efficient than reads that access remote or non-node local memory.

- The device uses a significant amount of memory-mapped I/O (MMIO) from the CPU directly to the device.

 For instance, some SCI interconnections use programmed I/O to memory-mapped I/O ports to pass low-latency messages between machines in a cluster.
- The actual code modules for the application and device driver are located on the node to which the I/O device is attached. I.e., the linear address of the code is physically mapped to memory on that node.
- The I/O buffers are located on the node to which the I/O device is attached.

To take advantage of these performance factors, the following practices are recommended:

- Locate a driver's specific I/O device in the system and allocate memory to that node.
- Specify interrupt (ISR and DPC) affinity to a specific CPU core and node and relocate driver code to the node where the driver, ISR, and DPC will execute.
- Stream data to buffers on the node where the I/O device is resident.

The conditions listed above increase performance primarily due to:

- Latency to/from the I/O device to the NUMA-closest memory, and CPU cache(s) is lower.
- OS scheduler opportunities—the code-flow sequence from application code to I/O API to kernel dispatcher to device driver code transition remains on one CPU core (for OSs that run a device from specific CPU cores) and is uninterrupted.
- The code fetched for the application and device driver is mapped to the closest memory. The coherent HyperTransport link no longer needs to fetch code blocks from far nodes, freeing coherent HyperTransport bandwidth for other traffic.

NUMA aware applications and drivers will ensure that your software will run with the highest performance possible across the many varying system topologies, from the single node desktop with a single noncoherent HyperTransport link to the eight socket server with multiple noncoherent HyperTransport links.

The discussion that follows makes frequent reference to Microsoft APIs. Developers creating applications to run under Linux™, Solaris or other operating systems should consult the NUMA API documentation specific to their target environment. The following recommendations are intended to compliment mechanisms provided by operating system to correctly establish the optimal device NUMA configuration.

E.3 Determining Number Of Nodes in AMD Family 10h and 12h Processor Systems in User-Mode

User mode programs can take advantage of the NUMA API support functions provided for various OSs to determine the number of nodes in the system, schedule threads, and allocate memory. While existing NUMA APIs provide traditional CPU-to-memory NUMA support, support functions to enumerate the system topology for I/O devices in a device-class independent manner are still under development.

A user-mode program must therefore create/allocate buffers and must depend to a large degree upon the operating system to allocate buffers optimally for the system into which the device is plugged. Currently most NUMA configuration information comes from the device-drivers and OS.

Applications should strive to allocate memory and schedule threads in a consistent manner (assigning threads to specific CPU(s) on a node) by using the NUMA API and allowing device drivers to do whatever is optimal—such as remapping linear to physical static I/O-buffers closer to the I/O device, copying buffers, etc., depending on the specific device and its latency characteristics. By grouping the threads that perform I/O to a specific core or cores on a node, it becomes easier to dynamically switch the entire device NUMA configuration to another CPU-cluster on another node to verify whether performance increases or decreases. However, the memory buffers do not “switch” automatically, even if the thread switches. The buffers remain on the nodes to which they were allocated and must be reallocated, if desired.

The examples that follow use Microsoft APIs. Developers who are creating applications running under Linux™, Solaris and other operating systems should consult the NUMA API documentation specific to their target environment. The guidelines in following section are recommended to compliment operating system mechanisms to correctly establish the optimal device NUMA configuration for a device.

E.4 Allocating I/O Device Buffers

Optimization

❖ Allocate I/O device buffers on the node where the operating system created the primary (`main`) thread.

Rationale

By default, the operating system may correctly create and run the primary thread on a CPU core on the node closest to the I/O device. By allocating all of the I/O device buffers from the same node, it then it becomes possible to dynamically reconfigure and optimize application performance.

Identify the CPU Core Running the Primary Thread

To determine the CPU core that is running the primary thread:

- On program entry call an operating system function such as `GetCurrentProcessorNumber()` from `Main()`:

  ```
  DWORD WINAPI GetCurrentProcessorNumber(void);
  ```

- Use the value returned to get the node by calling:

  ```
  GetNumaProcessorNode(currentProcessor, &currentNode);
  ```

E.5 I/O Thread Scheduling

Optimization

❖ Keep the I/O thread scheduled on the same node as the I/O device buffers.

Rationale

By assigning CPU affinity, threads can be maintained on the CPU closest to the I/O device buffers. Operating system functions, such as `VirtualAllocExNuma( )` can be used to specify the node to which to allocate the I/O device buffer-memory. The CPU affinity with the I/O threads to that node can be mapped using operating system functions, such as `SetThreadAffinityMask( )`.

E.6 Using Write-Only Buffers for Device Consumption

Optimization

❖ Use streaming-stores to optimize data for IO-device consumption, if the application is using a write-only buffer.

Rationale

The CPU writes to write-only output buffers, but does not read them. When using a write-only output buffer to be consumed by a device:

- Create the buffer on the node where the I/O device buffers are established.
- Use streaming-store instructions to avoid moving the buffer into the L1 cache of the writing CPU.
- Use the non-temporal streaming store instructions such as:

 MOVNTI—Stream to memory-integer

 MOVNTPS—Stream to memory-packed scalar floating-point

- Consult compiler intrinsic support to avoid assembly-language, such as:

```
void _mm_stream_ps(float * p , __m128 a ); // Uses MOVNTPS
```

When streaming the data by use of non-temporal instructions, data is write-combined on the node sending the data and then is forwarded to the node where the I/O-device buffer exists (see "Implementation of Write-Combining" on page 239). Streaming the data has two advantages:

- First, there is no coherent HyperTransport read traffic from memory into the L1 cache.
- Second the device read/write latency to the buffer can be lower if the buffer is closer to device.

E.7 Access of PCI Configuration Register

Optimization

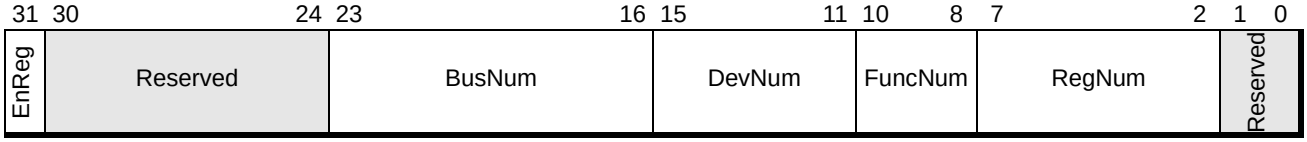
❖ Assure exclusive use of the PCI configuration address and data registers.

Rationale

Kernel mode drivers can use the operating system's low-level port access functions to read PCI configuration registers in the AMD64 CPU and integrated host bridge. These registers specify the system topology—the nodes on which each device resides.

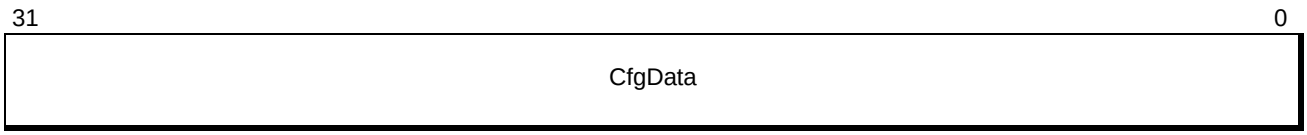
Unlike user mode, software running in kernel mode (such as a driver) can locate a specific I/O Device in an AMD64 system.

A brief description of how to generate PCI configuration space reads is described below. Consult the *BIOS and Kernel Developer's Guide for AMD Family 10h Processors, order# 31116,* for a more detailed description of PCI configuration space. The following scheme uses a configuration index port (configuration address register 0CF8h) as shown in Figure 13 and a configuration data port (configuration data register 0CFCh) as shown in Figure 14.



Bits	Mnemonic	Description	R/W
31	EnReg	Configuration Space Enable	R/W
30–24	Reserved	Reserved	
23–16	BusNum	Bus Number	R/W
15–11	DevNum	Device Number	R/W
10–8	FuncNum	Function Number	R/W
7–2	RegNum	Register Address	R/W
1–0	Reserved	Reserved	

Figure 13. Configuration Address Register (0CF8h)



Bits	Mnemonic	Description	R/W
31–0	CfgData	Configuration Data	R/W

Figure 14. Configuration Data Register (0CFCh)

Assuring Exclusive Use of Index and Data Ports

For thread-safety reasons, kernel mode function drivers should avoid performing the I/O directly to ports 0CF8h and 0CFCh in order to ensure that only a single thread is using the index and data ports exclusively.

For software performance testing, however, performing the I/O directly from a PCI-utility or debugger should not be problematic. A developer can use a PCI utility to enumerate the I/O topology and test performance, for example, of the I/O buffer allocations that are applicable, based on a given system's I/O topology. A PCI utility can also be useful for creating OEM system-specific topology for use by the I/O application/and or drivers.

The following pseudocode shows how (after ensuring exclusive access), an OS support routine or command-line debugger, performs I/O to read PCI configuration registers in the AMD CPU/Northbridge.

```
BUS 0x0, Device x18h (24), Function 0x1, register 0x60h // Node ID Register:

unsigned int busNum;
unsigned int devNum;
unsigned int funcNum;
unsigned int regNum;
unsigned int pci_registerSelect;
unsigned int pci_configData;


// Setup desired bus, device, and function number.

busNum=0x0;
devNum=0x18;
funcNum=0x1;
regNum=0x60;

// Setup the register with bus, device, and function.
// Also set the enable register read bit 31 0x80000000.

pci_register=(pci_registerSelect | 0x80000000);
pci_register=(pci_registerSelect | (busNum << 16));
pci_register=(pci_registerSelect | devNum << 11);
pci_register=(pci_registerSelect | (funcNum <<8));
pci_register=(pci_registerSelect | regNum);
```

```
// setup for PCI-configuration Read

#define PCI_CONFIGURATION_ADRESS 0xCF8
#define PCI_CONFIGURATION_DATA   0xCFC

__asm{
   mov  edx, PCI_CONFIGURATION_ADRESS;
   mov  eax, pci_registerSelect;
   out  edx, eax; //32-bit write
   mov  edx, PCI_CONFIGURATION_DATA
   in   eax, edx// read data
   mov  pci_configData, eax
       } // endasm here
```

When the system is operating under the runtime environment, the ideal thread-safe method by which to access the device-specific PCI configuration space is to use the operating system's PCI-bus driver. The methods to do this depend on the operating system.

E.8 Determine the Nodes that Have Noncoherent HyperTransport™ I/O Links

Optimization

❖ Inspect HyperTransport links on all nodes to whether it is both connected and configured as a noncoherent HyperTransport link.

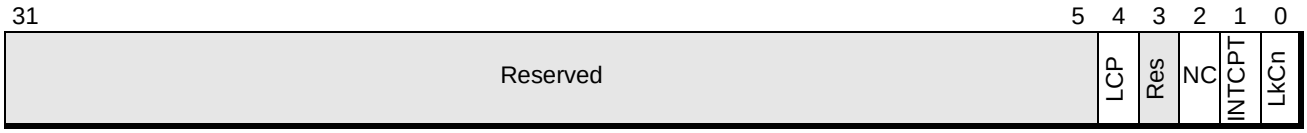
Rationale

After determining the system node count, software can identify the nodes attached to noncoherent HyperTransport links. If any link on the node is configured as a noncoherent HyperTransport link, record that node's number.

There are several ways to do this. For instance, in the following sections there are examples of how to associate an I/O device to a node by using the device's known PCI Bus number and by using the memory-mapped I/O address of the device that is provided in the device’s base address register (BAR).

Systems can have up to eight nodes. Every I/O-device in the system must be connected to one or more of nodes. Each node (cluster of CPUs) can be identified in the PCI configuration space using the PCI device ID on PCI bus 0 starting with device 24 (18h) function 0h and counting up to device 31 (1Fh) function 0h. Systems can have up to eight nodes; each node appears as one PCI device. Nodes with noncoherent HyperTransport links can be identified by reading the initialized values of the link connected bit (bit 0) and the noncoherent bit (bit 2) of the HyperTransport link type register for each HyperTransport link. There is one link type register for each link. Opteron supports up to three Hypertransport links; for each node, the registers are located at function 0h, registers 98h, B8h, D8h, F8h (links 0 through 3, respectively). The layout of the link type registers is shown in Figure 15.

2



Bits	Mnemonic	Description	R/W
31–5	Reserved		
4	LCP	Link Connect Pending	RO
3	Reserved		
2	NC	Noncoherent	RO
1	INTCPT	Initialization Complete	RO
0	LkCn	Link Connected	RO

Figure 15. Link Type Registers F0x[F8, D8, B8, 98]

Only One Node in System Has Noncoherent HyperTransport I/O Links

If only one node in the system contains noncoherent HyperTransport link (or links), then record that node number; this node contains all I/O-Devices in the system. Memory buffers that are accessed by the I/O device(s) for DMA should be allocated on this node for lowest device-to-memory latency. Optimally, the I/O driver code should also run on a CPU core on this node, resulting in the lowest latency for MMIO writes/reads to the device.

More Than One Node in System Has Noncoherent HyperTransport I/O Links

If more than one node contains a noncoherent HyperTransport link (or links), then the driver will need to associate the specific I/O device to a specific node in the system. This can be done using either of the following methods:

- Using the device's PCI-configuration base address register (BAR), software determines the node to which the I/O-Device is attached by comparing its base address to the contents of the MMIO routing table.
- Using the device's bus-number, the software obtains the node to which the I/O device is attached, based on the PCI bus to which PCI configuration cycles are steered.

Be sure to check the operating-system's API for other possible methods as well.

Determining the Location of the I/O-Device Using PCI-Configuration Base Address Registers

The first method uses the MMIO base and limit address registers (See Figure 16 and Figure 17 on page 301) to determine the location of the I/O device.

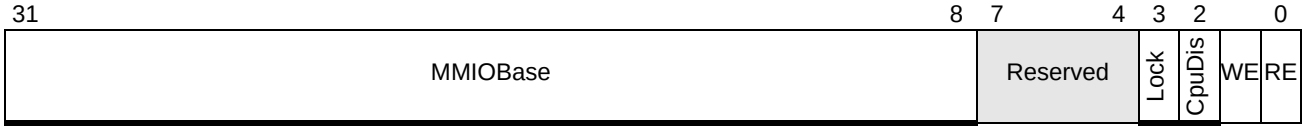
PCI/PCI-X/PCIe devices have a base address register (BAR) that allows the system BIOS and OS to map the device into the system address space. The BAR is either a 64-bit octword-aligned address in

the eight bytes starting at offset 10h, or a 32-bit octword-aligned address in the four bytes starting at offset 10h. Bits 2:1 of the byte at offset 10h distinguish the size of BAR as follows:

Table 18. Size of Base Address Register

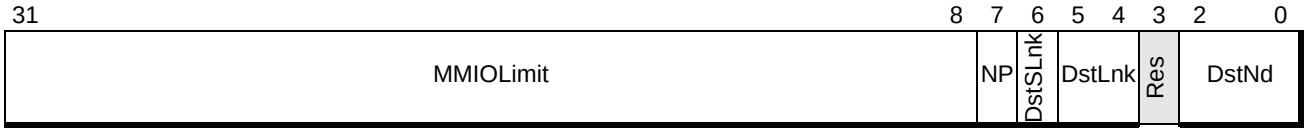
Value	Base Address Register Size	Device Location
00	32-bit decoder	Device is located anywhere in lower 4GB address space.
01	32-bit decoder	Device is located in lower 1MB address space.
10	64-bit decoder	Device is located anywhere in 2^{64} address space.
11	32-bit decoder	Reserved.

Most hardware is designed so that the BAR register contents can be read using MMIO or PCI configuration reads, if the driver can obtain the bus, device, and function of its device. Driver-code reads the BAR field and uses the address to find the node in the system that acts as the device bridge to decode this address in order to steer MMIO cycles into the appropriate down-stream bus (PCI, PCIe, PCI-X).



Bits	Mnemonic	Description	R/W
31–8	MMIOBase	MMIO base address register	R/W
7–4	Reserved		
3	Lock	Lock	R/O
2	CpuDis	CPU Disable	R/W
1	WE	Write Enable	R/W
0	RE	Read Enable	R/W

Figure 16. MMIO Base Address Registers F1x(B8h, B0h, A8h, A0h, 98h, 90h, 88h, 80h)



Bits	Mnemonic	Description	R/W
31–8	MMIOLimit	MMIO limit address register bits	R/W
7	NP	Non-posted	R/W
6	DstSLnk	Destination Sublink	R/W
5–4	DstLnk	Destination Link ID	R/W
3	Reserved		R/W
2–0	DstNd	Destination Node ID	R/W

Figure 17. MMIO Limit Address Registers F1x[BC, B4, AC, A4, 9C, 94, 8C, 84]

The following procedure uses the physical address of the device as an inclusive field to find the correct node hosting the device. This procedure compares addresses at a granularity of 64K, which is the granularity of allocation for MMIO regions. The offset is relative to bus 0, device 24, Function 1.

Step 1. Create 64-bit or 32-bit integer from the BAR Register based on bits 2:1.

Mask off bottom 4 bits (3:0) to create INT64 deviceBaseAddress; 0x00000000A0000000h.

Step 2. Select any (present in the system) node (node 0 etc), and check each of the 8 memory-mapped I/O address map registers to determine the node that will decode the I/O-Device. The memory-mapped I/O address map registers decode MMIO, if the physical address is less than or equal to, but not greater than, the limit field programmed into these registers.

Step 3. Initialize loop counter to 8.

Step 4. Move 0x80h to start_index (Bus 0, Device N (N=AMD Node Number, 18h, 19h, etc), Function 1, Register 0x80).

Step 5. Read PCI configuration register start_index into variable (int64) nodeMMIOaddress.

Step 6. Zero nodeMMIOaddress bottom 8 bits 7:0.

Step 7. Left-shift nodeMMIOaddress by 8.

Step 8. Now get node's MMIO limit by reading register start_index + 4 into int64 variable mmioLimit. Save a copy of mmioLimit to temporary variable (int64) orgMMIOlimit.

Step 9. Zero mmioLimit bottom 8 bits 7:0.

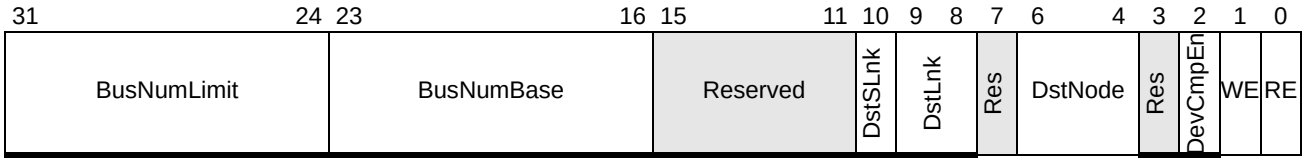
Step 10. Left-shift mmioLimit value by 8, then OR in bottom 16-bits to all '1's (0xFFFF). This is because the hardware effectively uses all '1's in lower 16-bits of the address.

Step 11. Compare deviceBaseAddress, to see if it is greater than or equal to nodeMMIOaddress. If so, we then need to check that deviceBaseAddress falls within the MMIO limit of the node, do this by checking that deviceBaseAddress is lower than or equal to mmioLimit. If this is true, we're done, save the node ID number by extracting bits 2:0 in orgMMIOlimit.

Step 12. If Step 12 did not provide match, increment start_index by 8 (i.e. 80h becomes 88h, which is the next Memory-Mapped I/O Base register), decrement loop counter and loop back to Step 5.

Determining Where the I/O-Device Is Located Using PCI-Bus Number of Device

Some operating systems provide a way for the driver to get system resources assigned to the device by use of a resource descriptor. For example Windows kernel-mode drivers can use the CM_PARTIAL_RESOURCE_DESCRIPTOR to obtain the bus number, interrupt-pin assignment, etc. Based on this information, the device node can be determined by querying the configuration map registers (see Figure 18 on page 303) to find the node into which the device is plugged.



Bits	Mnemonic	Description	R/W
31–24	BusNumLimit	Bus Number Limit	R/W
23–16	BusNumBase	Bus Number Base	R/W
15–11		Reserved	
10	DstSLnk	Destination Sublink.	R/W
9–8	DstLnk	Destination Link ID	W/R
7		Reserved	
6–4	DstNode	VGA enable	R/W
3		Reserved	
2	DevCmpEn	Device Number Compare Mode Enable	R/W
1	WE	Write Enable	R/W
0	RE	:Read Enable	R/W

Figure 18. Configuration Map Registers F1x[E0h, E4h, E8h, ECh]

First, get the PCI bus number of the device.

Step 1. Get Bus number. Create 32-bit integer. Save bus as `DevBusNum`.

Step 2. Check the bus number against the contents of the four configuration base/limit registers to determine the node with which the bus is associated. These registers reside in the Northbridge PCI configuration space at offsets E0h through ECh. Each register maps a contiguous range of PCI bus numbers to a particular node, allowing up to four separate bus ranges to be mapped to as many as four different nodes. The registers are replicated on each node so is it necessary to check the registers of only one node.

The remaining steps use the configuration map registers (Figure 18) to determine the node to which the device is attached.

Step 3. Read Northbridge PCI configuration space offset E0h and, test the device compare mode enable bit (bit 2 DevCmpEn).

Step 4. If bit is set, use the PCI bus Device number in place of the PCI bus number for this check (save device number as `DevBusNum`).

Step 5. Read each configuration base/limit register and compare `DevBusNum` against the range defined by the `BusNumBase` and `BusNumLimit` fields until the desired bus range is found.

Step 6. Get the node number from the `DstNode` field of that register. The IO device is attached to this node and the device's buffers should be allocated on this node. The range comparison succeeds if `DevBusNum` is greater than or equal to `BusNumBase` and less than or equal to `BusNumLimit`.

E.9 Using Interrupt Affinity

Optimization

❖ Make sure that interrupts from a device are serviced by a processor core that is on the same node as the device.

Rationale

Interrupt affinity is maintained by assuring that interrupt service routines (ISRs) from a device are run on a core that is on the same node as the device. There are various ways this enhances performance:

- Interrupt affinity can improve the service routine's cache locality—code and data have a better chance of being resident in the processor's L1, L2, or L3 cache. (L3 cache applies to 10h only)
- Handling the interrupt on the node where the device is located lowers the latency (number of hops) for any MMIO reads or writes that the service routine may make to the device.
- Performance can be further increased if memory, including device buffers, that the service routine accesses are allocated on the same node.

Setting up Interrupt Affinity

There are several ways to set up interrupt affinity.

- Use an OS-provided API to specify which CPU cores should run the interrupt service routine (ISR). Some OSs can allow the device's ISR to run on a set of primary CPUs. Select the cores on the node to which the I/O device is connected. Under Windows the driver can use the `WdfInterruptSetPolicy( )` function in conjunction with the `IoConnectInterrrupt( )` function to accomplish this (consult Microsoft driver development documentation for full details). Device buffers that the ISR accesses should be allocated on this node as well.
- Devices that are message-signaled interrupt (MSI or MSI-X) capable can specify interrupt affinity in the MSI message. Specify CPU cores on the node into which the I/O device is plugged. Message-signaled interrupts offer many performance improvements over legacy PCI/PCI-X line-based interrupts. Less interrupt sharing occurs, which decreases the latency required to service the interrupt.
- Specify which CPU cores should run the deferred procedure call (DPC) for further processing. If possible, queue the DPC that will be run after the ISR on the node into which the I/O-Device is plugged. Map the buffers used by the DPC to the node closest to the I/O-Device. For example, the Microsoft Windows operating system provides one DPC queue for each processor. Drivers can control the queue to which the operating system assigns the DPC. By default, when the driver calls `KeInsertQueueDpc( )` or `IoRequestDpc( )`, the DPC is queued on the currently active processor. In addition, drivers can specify the processor queue by calling `KeSetTargetProcessorDpc( )` before calling `KeInsertQueueDpc( )` or `IoRequestDpc( )`.

Appendix F Remarks on the RDTSC(P) Instruction

The RDTSC and RDTSCP instructions are used to load the value of the time stamp counter (TSC) into the EDX:EAX register pair. These two instructions differ as follows: the RDTSC instruction may execute speculatively and out of order with respect to other instructions (except other RDTSC instructions), while the RDTSCP instruction does not. The RDTSCP also identifies the processor core on which it is executed. When a code sequence ending in an RDTSC instruction is executed, there is no guarantee that all prior instructions in the code sequence have been retired at the time when the TSC is read.

On the other hand, the RDTSCP instruction waits for all the previous instructions to be retired before reading the TSC, thus producing the expected TSC value. For this reason, it is recommended that RDTSCP be used to measure the clock cycles consumed by a hotspot function. Both RDTSC and RDTSCP are executed in program order with respect to other RDTSC(P) instructions.

If RDTSC is used, it should be accompanied by a separate serializing instruction (such as a CPUID instruction). In AMD Family 10h and 12h processors, the MFENCE instruction, which is not intercepted in virtualized environments, can be used in place of the CPUID instruction as a serializing instruction.

In the previous generation multi-core processors, each core has its own timestamp counter locked to its core. Starting with AMD Family 10h and 12h processors, there exists a single clock source in the NorthBridge for all timestamp counters in a processor and these counters are incremented in lockstep. This enables the cycle counter to provide monotonically increasing values at a constant rate even when the cores are in power saving modes. This behavior of RDTSC(P) is indicated if EDX bit 8 is set to 1, as returned by CPUID function 8000_0007h. Note that an operating system can write different values to each core's TSC and can establish or correct a core-to-core skew, after which the TSCs all advance in lockstep with each other and thus maintain a constant core-to-core skew. The precision with which software can synchronize the TSCs across cores is dependent on the approach used, as well as platform factors, such as the consistency of inter-core communication latency through shared memory. This precision is typically limited to a few tens of cycles. In particular, the skew may exceed inter-core communication latencies such that inter-core observation of TSC values may not show strict monotonic behavior—a TSC value acquired from another core after the local TSC is read may have a lower value. Successive TSC reads within a core, however, give monotonically increasing values.

The HWCR[TscFreqSel] bit is set by the BIOS to scale the TSC frequency to the P(0) frequency of the CPU. To calculate the elapsed wall clock time from the values returned by two RDTSC(P) instructions, use the following formula.

Elapsed Wall Clock Time (in seconds) = (Second RDTSC result – First RDTSC result) / CPU's P(0) frequency.

The following example explains how to use the RDTSC(P) instruction to measure the clock cycles consumed by a hot spot function:

```
unsigned long long RDTSCP()
{
  unsigned long long tsc;
  asm volatile (".byte 0x0f, 0x01, 0xf9" : "=A" (tsc) : : "%ecx");
  return tsc;
}

#define MFENCE() asm volatile("mfence")

unsigned long long compute_mfence_overhead(int N)
{
  int i;
  unsigned long long tsc, next_tsc;
  tsc = RDTSCP();
  for (i = 0; i < N; i++)
    MFENCE();
  next_tsc = RDTSCP();
  return ((next_tsc - tsc)/N);
}

unsigned long long time_hotspot(int N)
{
  unsigned long long tsc, next_tsc, avg_hotspot_time;
  int i;

  /* start the timer */
  tsc = RDTSCP();
  /* N is the number of iterations. The higher the value of N,
     the more accurate the avg_hotspot_time (except for OS context switches.*/
  for (i = 0; i < N; i++)
    {
              /* MFENCE is used to serialize the control flow
                 between iterations.  */
      MFENCE();
      hotspot(); /* HotSpot function */
    }
  next_tsc = RDTSCP();

  avg_hotspot_time = (next_tsc - tsc)/N - compute_mfence_overhead(N);
  return (avg_hotspot_time);
}
```

Appendix G Guide to Instruction-Based Sampling on AMD Family 10h and 12h Processors

Instruction-Based Sampling (IBS) is a performance monitoring technique that provides precise information about AMD64 instruction fetch behavior and about the execution of operations that are issued from AMD64 instructions. This information can be used to analyze and improve the performance of programs executing on AMD Family 10h and 12h Processors.

IBS provides four important advantages over conventional performance counter sampling:

- Hardware events are attributed precisely to the instructions that cause the events. Conventional performance counter sampling is not precise, making it difficult, if not impossible, to attribute events to specific instructions. This limits the ability to pin-point performance issues at the instruction and source code levels.
- A wide range of events are monitored and collected with each IBS sample. Either multiple sampling runs or counter multiplexing must be used to collect the same range of information with conventional performance counter sampling.
- The virtual and physical addresses of load/store operands are collected. Profiling tools can use this information to associate specific data structures with the x86 instructions performing load/store operations.
- Latency is measured for key performance parameters such as data cache miss latency.

The precision afforded by IBS also enables automated optimization techniques (e.g., profile-directed optimization) which require detailed, precise information about instruction-level program behavior.

G.1 Background

Some familiarity with the microarchitecture of AMD Family 10h and 12h processors is required to understand how instruction-based sampling works and to interpret the data produced by IBS. Important information on the microarchitecture of AMD Family 10h and 12h Processors can be found in Appendix A, “Microarchitecture of AMD Family 10h and 12h Processors,” on page 223 of this volume. This section summarizes a few important points.

The *BIOS and Kernel Developers Guide (BKDG) for AMD Family 10h Processors*," order# 31116 and the *BIOS and Kernel Developers Guide for AMD Family 14h Processors*, order# 41131 provide many specific details about IBS (events, model specific registers, etc.) This appendix is intended to

complement the information in the BKDG. The BKDG should be regarded as the definitive resource about IBS features.

G.2 Overview

The successive pipeline phases of fetch, decode, dispatch and execution decouple the fetching of AMD64 instructions from their eventual conversion and execution as macro-ops. The separation between fetch and execution is reflected in the IBS mechanism. Instruction-based sampling consists of two parts:

- ***Fetch sampling***—collects and reports performance data on AMD64 instruction fetch behavior. Fetch sampling provides information about instruction TLB and instruction cache behavior for fetched instruction bytes.
- ***Op sampling***—collects and reports performance data on the execution of instruction operations (ops). Op sampling produces retirement cycle counts that are common to all sampled operations and execution-related data that are specific to the kind of operation (e.g., branch or return) that was sampled, including load and store operations.

A *fetch* is an access to the instruction cache that results in bytes being delivered to the decode instruction buffer and can contain multiple instructions.

Op sampling records and reports the address of the AMD64 instruction from which the op was generated and issued. This allows profilers and other supporting software tools to associate op performance data with a “parent” AMD64 instruction.

Fetch sampling and op sampling are independent and may be separately enabled. Fetch and op sampling may also be enabled at the same time. When fetch and op sampling are enabled at the same time, some additional interference will result due to the larger number of samples taken, effect on pipeline behavior, cache effects, etc. This behavior is a natural consequence of sampling. Fetch and op sampling each have their own model specific registers (MSRs) to control sampling and to report results.

The same overall process is used to take a fetch sample or an op sample. Generically, an IBS sample is taken in the following way:

1. A maximum instruction fetch (or op) selection count is loaded into the appropriate IBS control MSR.
2. IBS mode is enabled in the appropriate control MSR.
3. The periodic selection counter is incremented. For fetch sampling, the periodic fetch counter is incremented for each completed fetch. For op sampling, the periodic op counter is incremented each processor cycle (cycles-based selection mode) or the periodic op counter is incremented each time an op is dispatched (dispatched op-based selection).
4. When the selection counter reaches the maximum selection count, an instruction fetch (or op) is selected and the instruction fetch (or op) is tagged.

5. As the tagged instruction fetch (or op) is processed by the hardware, events that occur due to the tagged instruction fetch (or tagged op) are recorded by the hardware (e.g., did it cause a cache miss, branch mispredict, etc.).
6. If the tagged instruction fetch (or tagged op) finishes, an interrupt is raised and all of the collected information is passed to an interrupt service routine (ISR) through the IBS MSRs.
7. The ISR saves the information producing an IBS sample. After collecting the sample, the ISR clears the current count and goes to step 2.

Software using IBS needs to take a few differences between fetch and op sampling into account in order to handle both kinds of performance data. The differences between fetch and op sampling are described in the following sections.

G.3 IBS fetch sampling

This section describes IBS fetch sampling in more depth. IBS fetch sampling captures information about the process of fetching instruction bytes.

An “attempted fetch” is a request to load instruction bytes from a specific virtual memory address. (The term “linear” is sometimes used in place of “virtual” when referring to virtual addresses.) A “completed fetch” is an attempted fetch (a request) that eventually delivers instruction bytes to the decoders. An “aborted fetch” is a request that does not complete.

G.3.1 Taking an IBS fetch sample

IBS fetch sampling is controlled by values that are configured into specific IBS MSRs. (See the BKDG for details.) The periodic fetch counter and the maximum fetch count value control the sampling process. The IBS fetch sampling mechanism counts completed fetches in the periodic fetch counter in order to select the next attempted fetch to tag and monitor. When the periodic fetch counter reaches the maximum fetch count, the next attempted fetch is tagged and monitored. Thus, the maximum fetch select count determines how often an attempted fetch is selected and tagged. This quantity is often called the “sampling period.” An interrupt is generated when the tagged fetch completes or aborts. IBS information is reported for both completed and aborted fetches.

Both the periodic fetch counter and the maximum fetch count are 20-bit values. Software can set the high order 16 bits of the maximum fetch count; the low order 4 bits are always set to zero. The periodic fetch counter is reset when the maximum fetch count is reached. Under software control, the low order 4 bits of the periodic fetch counter can be set to a pseudo-random 4-bit value. Randomization helps to prevent the sampling process from “syncing up” with tight loops that are being executed—a periodicity effect that could affect the accuracy of performance statistics. Software may also choose to randomize the maximum fetch count by generating its own randomized maximum fetch count.

IBS fetch sampling data are returned in MSRs that are read by the interrupt service routine. The following information about the tagged fetch is returned:

- The virtual fetch address (always valid),
- The corresponding physical fetch address (valid when the MSR flag IbsPhysAddrValid is set),
- Event flags indicating completion, IC miss, L1 ITLB miss or L2 ITLB miss,
- Size of the page translation in the L1 ITLB (a 2-bit field that is valid when the MSR flag IbsPhysAddrValid is set), and
- The instruction fetch latency in cycles.

Software may augment the IBS fetch data with other information such as a timestamp, the process identifier, etc. The virtual fetch address directly relates the information in the sample to the fetched instruction bytes—something which cannot be done with counter sampling since conventional counter sampling records the interrupt restart IP instead of the instruction that actually caused the events. Out-of-order execution further complicates the attribution of an IP to events since many instructions may be in-flight. The IBS virtual and physical fetch addresses precisely identify the instruction that caused the events reported by the hardware—the key advantage of IBS.

The following table summarizes the hardware event flags and values that are available for analysis.

Table 19. IBS Hardware Event Flags

Flag/Field	Purpose/meaning
IbsPhyAddrValid	Physical address and page size are valid
IbsL1TlbMiss	Fetch initially missed in L1 ITLB
IbsL2TlbMiss	Fetch initially missed in L2 ITLB
IbsL1TlbPgSz	Size of page translation in L1 ITLB
IbsIcMiss	Fetch initially missed in the instruction cache
IbsFetchComp	Fetch completed
IbsFetchLat	Cycles from fetch initiated to completed/aborted

The flag/field names used in this table correspond to the flags and fields in the IBS model-specific registers. Please see the *BIOS and Kernel Developer's Guide for AMD Family 10h Processors*, order# 31116 or the *BIOS and Kernel Developer's Guide for AMD Family 14h Processors*, order# 41131 for the location, size, and position of these flags and fields.

The IBS fetch latency value (IbsFetchLat) reports the number of cycles between the instant the fetch was initiated and the instant an instruction was delivered to the decoder (completion) or until the instant when the fetch was aborted. The translation page size (IbsL1TlbPgSz) is a 2-bit field that indicates the size of the page that was used during virtual to physical address translation as performed in the L1 ITLB.

G.3.2 Interpreting IBS fetch data

Before discussing the interpretation of IBS fetch data, it helps to have a little background on the process of fetching instruction bytes.

An attempted fetch must first be processed by the instruction translation lookaside buffers (ITLB) to convert the virtual address to a physical address. AMD Family 10h and 12h Processors use a two-level ITLB structure consisting of a level 1 (L1) ITLB and a level 2 (L2) ITLB. If page translation information is found in the L1 ITLB, the virtual address is translated using that information. If translation information is not in the L1 ITLB (an L1 miss) and it is available in the L2 ITLB, the L1 ITLB is reloaded with the information from the L2 ITLB and address translation completes. If neither ITLB contains the page translation information (an L1 ITLB miss and an L2 ITLB miss), then the information is loaded from memory-resident page tables.

Once the physical address is available, the instruction cache (IC) is accessed using the physical address. If the instruction bytes for that address are present in the IC, it is returned and is delivered to the decoders. If the instruction bytes are not present in the IC (an IC miss), it must be obtained from either the L2 cache, L3 cache, or system memory. In any case, the instruction bytes are delivered to the decoders once it is available.

As stated earlier, an attempted fetch is said to complete when its instruction bytes are delivered to the decoders. An attempted fetch that did not complete is an aborted fetch. An attempted fetch may abort at any point in the process of fetching instruction bytes. A fetch may abort due to a control transfer misprediction from an earlier fetch.

Instruction fetch is a highly speculative activity. Some completed fetches could also be on the wrong path, but the redirection does not arrive until after the instruction has left the IC. Thus, some completed fetches are speculative and the corresponding instructions may not be executed or retired.

The hardware event flags and values in the previous table are made available to software when an IBS fetch sampling interrupt is generated. The miss event flags indicate whether the attempted fetch initially missed the L1 ITLB, L2 ITLB, or IC. Address translation and IC access may eventually succeed. These flags show the hardware condition on the first attempt at translation/access. Eight combinations of the event flags, as shown in the table below, are produced by the hardware.

Table 20. Event Flag Combinations

TlbMiss	IbsPhyAddrValid	IbsFetchComp	IbsIcMiss	Interpetation
0	0	0	0	Killed by redirect before ITLB/IC access
0	1	0	1	L1 ITLB hit, IC miss, likely redirect during IC fill
0	1	1	0	L1 ITLB hit, IC hit
0	1	1	1	L1 ITLB hit, IC miss
1	0	0	0	ITLB miss, likely redirect during reload of L1 ITLB
1	1	0	1	ITLB miss, IC miss, likely redirect during IC fill
1	1	1	0	ITLB miss, IC hit
1	1	1	1	ITLB miss, IC miss

In Table 20, “TlbMiss” means “IbsL1TlbMiss or IbsL2TlbMiss”; the logical OR of the IBS event flags IbsL1TlbMiss and IbsL2TlbMiss.

The first case, in which IbsL1TlbMiss, IbsL2TlbMiss, IbsPhyAddrValid, IbsFetchComp, and IbsIcMiss are all clear, does not provide any useful information since the attempted fetch is killed very early before ITLB or IC access. We refer to such fetches as “killed fetches.”

The other three cases in which a fetch does not complete are also likely due to a redirection. These four cases are the result of incorrect branch speculation. While completed fetches have “full” information, the IC and ITLB information for aborted fetches is just as important. Instruction cache and TLB accesses can be both constructively and destructively influenced by earlier wrong path accesses.

G.4 IBS op sampling

This section describes IBS op sampling in more detail. IBS op sampling provides information about the execution of ops.

G.4.1 Taking an IBS op sample

IBS op sampling is controlled by values that are configured into specific IBS MSRs (refer to the BKDG for details). The periodic op selection counter and the maximum count value control the sampling process. The current periodic op selection count is maintained in a hardware counter. Software can read and write this counter in order to save and restore its current value, and to preset the count. The maximum op selection count value determines how often a micro-op is sampled (the sampling period).

On AMD Family 10h processors, the current periodic op selection count and the maximum count value are each 20 bits. The high order 16 bits of the 20-bit maximum count value can be configured by software. The internal, low order 4 bits of the maximum op selection count are always zero.

On AMD Family 12h processors, the current periodic op selection count and the maximum count value are each 27 bits. The high order 23 bits of the 27-bit maximum count value can be configured by software. The internal, low order 4 bits of the maximum op selection count are always zero.

IBS provides two op selection and tagging modes: cycles-based selection and dispatch op-based selection. The mode is selected by software through the IBS op control MSR.

- **Cycles-based selection**. IBS counts processor cycles in order to select and tag an op for sampling. The current periodic op selection count is incremented each processor cycle. When the current count reaches the maximum op selection value, a one-of-four round-robin counter selects an op in the next dispatch line. If the op selected by the round-robin counter is invalid, the next younger op is tagged.
- **Dispatched op-based selection**. IBS counts dispatched ops in order to select and tag an op for sampling. When the current periodic op selection count reaches the maximum op selection value,

the op is tagged. Dispatched op-based selection is available on AMD Family 10h RevC (and later) processors, and on AMD Family 12h processors.

In both modes, an interrupt is generated when a tagged op is retired. The IBS event and latency values are then read from the MSRs by the interrupt service routine. The interrupt service routine may combine the IBS op data with other information (such as a timestamp and process ID) forming a complete software sample to be saved for post-processing.

Software must randomize the sampling period. Coincident periodicity may occur between the sampling process and the workload. Periodicity affects op selection, producing profiles that exhibit aliasing effects (over- or under-sampling of certain ops/instructions). Randomization of the low order 4 bits is especially important. This must be accomplished by writing a pseudo-random value to the periodic op selection counter since the low order 4 bits of the maximum op selection count are always zero.

Ops may stall in the pipeline stage in which they are tagged. When an op is stalled, it is more likely to be tagged. This behavior affects the statistical distribution of ops in the resulting program profile.

IBS only returns data for tagged ops that retire. However, a tagged op may be flushed before retirement. In this case, IBS data for the flushed op is discarded (i.e., the sample is dropped.). The number of dropped samples due to a flushed tagged op can be counted by a performance monitoring event (see BKDG). After a tagged op is flushed, the current count is set to a pseudo-randomized 7-bit value and a new op is tagged when the current count again reaches the maximum count value.

G.4.2 Interpreting IBS op data

An op can be classified into one of several broad categories according to the major operation which it performs: arithmetic, logic, shift, etc. In general, IBS treats ops as *undifferentiated*, that is, the category or function is not explicitly identified by IBS. However, two categories of ops are explicitly identified: *branch* and *resync*. A branch op implements AMD64 branch semantics and includes unconditional jumps, conditional jumps, subroutine call and return. One subtype of branch op is also explicitly identified by IBS: *return*. A return op implements AMD64 return semantics. A resync op is only found in certain micro-coded instructions and causes a complete pipeline flush. Branch, return and resync are explicitly identified since interesting information about program control flow can be obtained by monitoring their behavior.

In addition to performing a major function (such as arithmetic, branch, etc.), an op may initiate a memory read, memory write, or a read and write to the same memory address. IBS explicitly identifies those ops which perform a *load* (memory read) and/or *store* (memory write) operation. When interpreting IBS data, please note:

- Some ops can perform a “load-operate-store” sequence to the same address and are identified by IBS as performing both a load *and* a store operation.
- Some branch ops perform a load operation and will be identified by IBS as performing a load.

The exact type of the sampled op is specified by one or more bits in the IBS MSRs that return sample information:

- The IbsOpBrnRet (where the “Ret” suffix stands for “retired”) and IbsOpReturn bits in the IbsOpData MSR indicate whether the op was a branch or return.
- The IbsOpBrnResync bit in the IbsOpData MSR indicates whether the op was a resync.

If none of these bits are set, the op is undifferentiated. Undifferentiated ops are still important as they provide information about program execution. The IbsLdOp and IbsStOp bits in the IbsOpData3 MSR indicate whether the op performed a load operation or a store operation, respectively.

Three values are reported for all ops:

- The virtual address of the parent AMD64 instruction from which the tagged op was issued (IbsOpRip)
- The tag-to-retire count in cycles (IbsTagToRetCtr), and
- The completion-to-retire count in cycles (IbsCompToRetCtr).

These values are returned in model-specific registers.

The virtual address of the parent AMD64 instruction can be used to associate the IBS op sample with the AMD64 instruction from which the op was issued. More than one op may be issued from a single AMD64 instruction. All such ops have the same virtual instruction address (RIP) as the parent AMD64 instruction.

Tag-to-retire count and completion-to-retire count are retire-related cycle counts.

Tag-to-retire Count

The tag-to-retire count is the number of cycles between the instant the op is tagged to the instant the op is retired. The op is tagged when it leaves the decode unit.

Instructions can stall after they are decoded due to a lack of resources, such as reservation station entries. These cycles are included in the tag-to-retire count.

The tag-to-retire time includes the time spent waiting for operands, time spent waiting to issue after operands are available, time spent in an execution unit, and time spent waiting for all the younger ops in the scheduling window to retire.

Completion-to-retire count

The completion-to-retire count is the number of cycles between the instant the op completed and the instant the op was retired. An operation is complete when it has finished execution. The completion-to-retire count indicates how long retirement was delayed after completion.

The difference between the completion-to-retire count and the tag-to-retire count is the number of cycles that occur between tagging and completion.

G.4.3 Interpreting IBS branch/return/resync op data

Information about branch, return and resync ops are reported in the the IbsOpData MSR. The event flags and counts returned by the IbsOpData MSR are summarized in the following table.

Table 21. IbsOpData MSR Event Flags and Counts

Flag/field	Purpose/meaning
IbsOpBrnRet	Op was a retired branch
IbsOpBrnMisp	Op was a branch that mispredicted
IbsOpBrnTaken	Op was a branch that was taken
IbsOpReturn	Op was a return
IbsOpMispReturn	Op was a return that mispredicted
IbsOpBrnResync	Op was a resync
IbsRipInvalid	RIP is not valid for the op (Family 12h only)
IbsTagToRetCtr	Cycles from op tagging to retirement
IbsCompToRetCtr	Cycles from op completion to retirement

As noted earlier, the IbsTagToRetCtr and IbsCompToRetCtr fields are valid for all op samples, not just branch, return and resync ops.

A branch operation is a change in program control flow (or micro-code control flow for IbsOpBrnMisp and IbsOpBrnTaken). Information is reported only for retired branches since IBS data is only reported for retired ops. (Information for flushed ops is not reported.) Mispredicted branches retire then kill all younger ops after them and redirect the front end of the pipeline.

The IbsOpBrnRet flag indicates whether the tagged op was an operation with AMD64 branch semantics (set) or not (clear.) If the IbsOpBrnFlag is set then the IbsOpBrnMisp and IbsOpBrnTaken flags indicate the execution status of the op in the following way:

Table 22. Execution Status Indicated by IbsOpBrnMisp and IbsOpBrnTaken Flags

IbsOpBrnMisp	IbsOpBrnTaken	Execution status
0	0	Was not mispredicted and was not taken
0	1	Was not mispredicted and was taken
1	0	Was mispredicted and was not taken
1	1	Was mispredicted and was taken

The IbsOpBrnMisp and IbsOpBrnTaken bits can be viewed as a property of the tagged branch op.

The IbsOpReturn and IbsOpMispReturn flags can also be considered to be properties of the tagged AMD64 or micro-coded branch op. The IbsOpReturn flag is set when the tagged op was, specifically, a return op. The IbsOpMispReturn flag indicates whether the return op was mispredicted or not. A resync op is not predicted and is always taken.

Table 23. Execution Status Indicated by IbsOpReturn and IbsOpMispReturn Flags

IbsOpReturn	IbsOpMispReturn	Execution status
0	N/A	Branch op was not a return
1	0	Was a correctly predicted return op
1	1	Was a mispredicted return op

IbsOpBrnMisp and IbsOpBrnTaken are valid for all branch ops including micro-code ops and return ops. The IbsOpReturn and IbsOpMispReturn flags merely provide additional information for return ops.

On AMD Family 12h processors, IBS reports the branch target address in a separate MSR. The branch target address is valid if it is non-zero. The target address may be used to build a dynamic control graph for frequently executed code, including control edges that cannot be determined through static analysis (for example, edges due to indirect jumps).

G.4.4 Interpreting IBS Load/Store Data

If the sampled op accesses memory, information about the load and/or store operation is returned in four model specific registers:

- The IbsOpData2 and IbsOpData3 registers contain event flag and latency information accumulated in the Northbridge and load/store unit, respectively.
- The IbsDcLinAd and IbsDcPhysAd registers contain the virtual (linear) and physical address of the memory operand, i.e., the address of the memory location read and/or written.

The address of the memory operand can be used to associate the load or store operation with a data structure in memory. The virtual and physical addresses are valid when the IbsDcLinAddrValid and IbsDcPhyAddrValid bits are set, respectively, in the IbsOpData3 register. Some ops that are issued from micro-coded instructions use a physical address directly. Thus, it is possible to have an IBS sample with a valid physical address and an *invalid* virtual address.

The flags and fields in the IbsOpData3 MSR provide basic information about any memory access initiated by the sampled op. The IbsLdOp and IbsStOp fields indicate whether a load and/or store were initiated by the sampled op. If a load operation initially misses in the data cache (as indicated by IbsDcMiss), the IbsDcMissLat field returns the number of clock cycles from when the miss was detected until data was delivered to the core. This field is not valid for store operations.

IbsDcStBnkCon and IbsDcLdBnkCon fields are set when a memory op cannot access the cache due to a bank conflict, resulting in a delay of the op.

The data cache miss latency (IbsDcMissLat Dcache) is only valid for *loads* that miss in the data cache. The timed latency interval for the IbsDcMissLat is is calculated from the data cache miss to data cache write.

The following table summarizes the information in the IbsOpData3 register.

Table 24. IbsOpData3 Register Information

Flag/field	Purpose/Meaning
IbsLdOp	Tagged op initiated a load operation
IbsStOp	Tagged op initiated a store operation
IbsDcL1tlbMiss	Translation info not initially present in L1 DTLB
IbsDcL2tlbMiss	Translation info not initially present in L2 DTLB
IbsDcL1tlbHit2M	Translation info was eventually present in a 2M page entry in L1 DTLB
IbsDcL1tlbHit1G	Translation info was eventually present in a 1G page entry in L1 DTLB
IbsDcL2tlbHit2M	Translation info was initially present in a 2M page entry in L2 DTLB
IbsDcMiss	Load/store initially missed in the data cache
IbsDcMisAcc	Load/store crossed a 128-bit address boundary (misaligned)
IbsDcLdBnkCon	Load/store had a bank conflict with a load.
IbsDcStBnkCon	Load/store had a bank conflict with a store.
IbsDcStToLdFwd	Data was forwarded from a store to the tagged load
IbsDcStToLdCan	Forwarding from a store to the tagged load was cancelled
IbsDcUcMemAcc	Load/store accessed uncacheable memory
IbsDcWcMemAcc	Load/store accessed write combining memory
IbsDcLockedOp	Load/store was a locked operation
IbsDcMabHit	Load/store hit on an allocated MAB entry
IbsDcLinAddrValid	Virtual (linear) address valid
IbsDcPhyAddrValid	Physical address valid
IbsDcMissLat	Data cache miss latency (load only)

G.4.5 Interpreting IBS load/store Northbridge data

The memory hierarchy in the AMD Family 10h and 12h Processors consist of separate L1 data and instruction caches, L2 cache, L3 cache (10h only) and system memory. Each core has its own L1 data and instruction caches and L2 cache. The L3 cache is shared among the cores within a multi-core processor. System memory is supported by a non-uniform memory access (NUMA) architecture in which some portion of physical memory is local to the processor while the remaining portions of physical memory are remote. Access to remote memory is implemented through the AMD Direct Connect Architecture via coherent HyperTransport™ links.

If a core cannot satisfy a load or store operation from L1 data cache or L2 cache, it communicates a request to the Northbridge through its System Request Interface. The Northbridge is shared across cores. The Northbridge consists of:

- A System Request Interface (SRI) to each core,
- A shared L3 cache (if present),
- A memory controller (MCT) to handle communication with local memory,

- One or more HyperTransport (HT) link interfaces, and
- A crossbar (XBAR) to handle communication between the SRI, MCT and HT links.

Further, the Northbridge performs address space routing. There are four main types of address space routing: system memory (DRAM), Memory-mapped IO (MMIO), IO space, and configuration space. (See the *BIOS and Kernel Developer's Guide for AMD Family 10h Processors*, *order# 31116*, for more information about this and other aspects of the Northbridge.) The Northbridge also handles communication between a core and its local Advanced Programmable Interrupt Controller (APIC.)

When the Northbridge receives a request for data through the SRI, the data will be retrieved from one of several data sources depending upon the physical location of the data and possibly its coherency state. Relative to the processor making the request, data may be returned from:

- L1 or L2 cache in one of the other local cores
- Local L3 cache
- Remote L1/L2/L3 cache (after traversing a coherent HT link)
- Local system memory (via the local MCT)
- Remote system memory (after traversing a coherent HT link and via the remote MCT)
- Local MMIO, configuration space, or APIC
- Remote MMIO or configuration space

Information about Northbridge activity is gathered and returned when IBS op sampling is enabled and a load operation misses in both the L1 data cache and the L2 cache. The IbsOpData2 register returns information from the Northbridge. Data in this register is valid when a load misses in both the L1 data cache and the L2 cache. The fields in the IbsOpData2 register are summarized in the following table.

Table 25. IbsOpData2 Register Fields

Flag/field	Purpose/Meaning
NbIbsReqCacheHitSt	Modified state (0), Owned state (1)
NbIbsReqDstProc	Request serviced by local (0) or remote (1) memory
NbIbsReqSrc	Data source (See table below)

Note that the NbIbsReqCacheHitSt and NbIbsReqDstProc fields are not supported on AMD Family 12h processors. It is important to emphasize that Northbridge data are only valid for load operations. Store operations may retire before they read data into the local cache. Thus, a subset of IBS information is either invalid or unreliable for store operations because a store operation may have retired and caused a sampling interrupt before store-related Northbridge events even occur. This behavior affects the validity of IBS data cache miss latency, which is only valid for load operations. Software developers should filter out NB and IBS data cache miss latency for store operations and report data only for load operations.

The NbIbsReqDstProc bit indicates whether the request was serviced locally or by a remote processor. Local service is typically faster. The NbIbsReqSrc field indicates the data source which satisfied the request, as described in Table 26. The NbIbsReqCacheHitSt indicates the cache state (modified or owned) when the data source type is “Cache.”

Table 26. Northbridge Request Data Source Field

NbIbsReqSrc	Northbridge Request Data Source
0x0	No valid status
0x1	Data returned from local L3 cache
0x2	Data returned from local CPU cache in another core or remote L1/L2/L3 cache
0x3	Data returned from DRAM
0x4	Reserved
0x5	Reserved
0x6	Reserved
0x7	Data returned from MMIO/configuration space/PCI/APIC

IBS Northbridge event data may be interpreted according to the following table.

Table 27. IBS Northbridge Event Data

NbIbsReqSrc	NbIbsReqDstProc	Meaning
0	0	Northbridge data is invalid
0	1	Northbridge data is invalid
1	0	Request served from local L3 cache
1	1	N/A
2	0	Request served from L1 or L2 of a local core
2	1	Request served from L1 or L2 of a remote core or a remote L3 cache
3	0	Request served from local DRAM
3	1	Request served from remote DRAM
7	0	Request served from local MMIO/Config/PCI/APIC
7	1	Request served from remote MMIO/Config/PCI

G.5 Software-based analysis

The approach to event computation and reporting described in this section is based upon concepts and methods that are used to compute and report results obtained through conventional performance counter sampling. This approach has two main advantages:

- Software developers and other end users are already familiar with performance counter sampling and the hardware events that are measured and reported.

- Profilers and other software tools can exploit existing data structures and methods to post-process IBS sample data and to correlate events with instructions, source lines, functions, modules, threads and processes.

The main disadvantage is that the full flexibility of analysis afforded by IBS data is not fully realized. Unfortunately, description of alternative approaches is beyond the scope of this paper.

G.5.1 Derived events and post-processing

The method described here converts IBS sample data into a set of derived event counts. A *derived event* is a specific, useful hardware condition that can be determined through a combination of one or more IBS event flags or values (such as the translation page size, tag-to-retire count, etc.) Example derived events include IBS instruction cache miss, IBS mispredicted branch and IBS data cache miss.

Caution: *End users should be discouraged from making direct comparison between IBS derived events and the performance counter events with the same or similar names because the sampling method and populations are different. Performance counter sampling is triggered by an event configured for a counter. IBS fetch sampling is triggered by completed fetches and IBS op sampling is triggered by processor cycles. Also, PMC execution events may be triggered by any op while IBS op events are counted only for retired ops.*

The process of converting IBS sample data to derived event counts is straightforward. An IBS fetch sample may be drawn as shown in :

Table 28. An IBS Fetch Sample

Virtual fetch address	IbsIcMiss	IbsL1TlbMiss	IbsL2TlbMiss	IbsComp	…
0x04000020	1	0		1	

Each column represents a hardware flag or value returned with an IBS sample. The virtual fetch address can be correlated back to an instruction in a software process using the same well-known techniques employed in conventional performance counter sampling. To visualize the process of computing derived events, consider the ten IBS fetch samples arranged as a 2-D table, where each row is a sample:

Table 29. 2-D Table of IBS Fetch Samples

Virtual fetch address	IbsIcMiss	IbsL1TlbMiss	IbsL2TlbMiss	IbsComp	…
0x0400020	1	1	0	1	
0x0400040	0	0	0	1	
0x0400020	0	0	0	1	
0x0400040	0	0	0	1	
0x0400020	0	0	0	0	

Table 29. 2-D Table of IBS Fetch Samples

0x0400040	0	0	0	1	
0x0400040	0	0	0	1	
0x0400020	0	0	0	1	
0x0400044	0	0	0	1	
0x0400080	1	0	0	1	

To compute the three derived events:

- IBS instruction cache miss
- IBS L1 ITLB miss, L2 ITLB hit
- IBS fetch completed

scan sequentially through the table from top to bottom and count the number of occurrences of the hardware conditions associated with the three events. (In practice, post-processing software must keep a running count for all derived events.) In the case of IBS instruction cache miss, for example, there are two samples with the IbsIcMiss = 1 condition, so the total count reported for this event is two. The number of IBS fetch completed events is nine.

The IBS L1 ITLB miss, L2 ITLB hit event requires the use of a slightly more complicated condition, (IbsL1TlbMiss & ~IbsL2TlbMiss), but the counting procedure is the same. The count reported for this event is one, since only one IBS fetch sample satisfies the condition.

Derived events can be placed into a histogram to obtain a program profile. The histogram for the IBS fetch completed derived event.

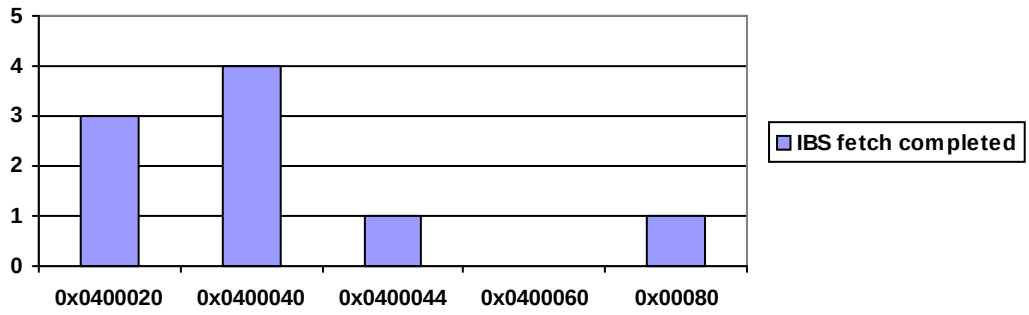


Figure 19. Histogram for the IBS Fetch Completed Derived Event

The following sections define derived events for IBS fetch and op data.

G.5.2 Derived events for IBS fetch data

Performance analysis tools (e.g., a profiler) can combine event flags to derive new events. Table 30 illustrates the kinds of events that can be derived from the basic flags and fields provided in an IBS fetch sample.

Table 30. New Events Derived from Combined Event Flags

ID	Name	Derivation
F000	IBS fetch samples	Number of all IBS fetch samples
F001	IBS fetch killed	Number of killed IBS fetch samples
F002	IBS fetch attempted	Number of non-killed IBS fetch samples
F003	IBS fetch completed	IbsFetchComp
F004	IBS fetch aborted	~IbsFetchComp
F005	IBS L1 ITLB hit	~IbsL1TlbMiss & IbsPhyAddrValid
F006	IBS L1 ITLB miss, L2 ITLB hit	IbsL1TlbMiss & ~IbsL2TlbMiss
F007	IBS L1 ITLB miss, L2 ITLB miss	IbsL1TlbMiss & IbsL2TlbMiss
F008	IBS instruction cache miss	IbsIcMiss
F009	IBS instruction cache hit	IbsFetchComp && ~IbsIcMiss
F00A	IBS 4K page translation	IbsL1TlbPgSz=0 & IbsPhyAddrValid
F00B	IBS 2M page translation	IbsL1TlbPgSz=1 & IbsPhyAddrValid
F00C	IBS 1G page translation	IbsL1TlbPgSz=2 & IbsPhyAddrValid
F00D	Reserved	
F00E	IBS fetch latency	IbsfetchLat

The ID numbers in Table 30 are the event identifiers that the AMD CodeAnalyst™ profiler uses to identify a derived event.

The first five derived events break down the IBS fetch samples by five broad categories.

- **IBS fetch samples** is the number of all IBS fetch samples that were taken.
- **IBS fetch killed** is the number of all IBS fetch samples that were killed before ITLB/IC access IbsL1TlbMiss, IbsL2TlbMiss, IbsPhyAddrValid, IbsFetchComp and IbsIcMiss are all clear.
- **IBS fetch attempted** is the number of IBS fetch samples minus the number of IBS fetch killed samples.
- **IBS fetch completed** is the number of attempted fetches that completed, i.e., delivered instruction bytes to the decoder.
- **IBS fetch aborted** is the number of attempted fetches that did not complete.

It should be noted that the notion of an attempted fetch here excludes killed fetches. Killed fetches do not provide useful analytical information and are filtered out. Killed fetches are not included in the IBS fetch aborted derived event.

For the remaining derived events, an occurrence of an event is tallied if it meets the specified condition in the Derivation column.

The IbsPhysAddrValid bit is needed to form the derived event “IBS L1 ITLB hit” because the condition (IbsL1TlbMiss=0 & IbsL2TlbMiss=0) alone is not sufficient to determine whether the attempted fetch hit in both the L1 and L2 ITLB or that the attempted fetch completed the initial ITLB access. When IbsPhysAddrValid is set, it indicates that address translation completed and produced a physical address. This bit must also be used to detect valid page translation information.

G.5.3 Derived Events for all Ops

A derived event is an event that is formed using a combination of IBS event flags and field values. The quantity of each derived event is computed and reported to end users by profiling software.

There are three derived events that are defined for all ops regardless of type:

Table 31. Derived Events for All Ops

ID	Name	Derivation
F100	IBS all op samples	Number of all IBS op samples
F101	IBS tag to retire cycles	Sum of all tag to retire cycles
F102	IBS completion to retire cycles	Sum of all completion to retire cycles

The **IBS all op samples** derived event is a count of all IBS op samples taken without regard to op type (i.e., undifferentiated, branch/return/resync.) Similarly, the **IBS tag to retire cycles** and **IBS completion to retire cycles** are computed across all IBS op samples for a given IP.

G.5.4 Derived events for IBS branch/return/resync ops

The following derived events measure the behavior of branch, return and resync ops:

Table 32. Derived Events to Measure Branch, Return and Resync Ops

ID	Name	Derivation
F103	IBS-branch op	IbsOpBrnRet
F104	IBS mispredicted branch op	IbsOpBrnRet & IbsOpBrnMisp
F105	IBS taken branch op	IbsOpBrnRet & IbsOpBrnTaken
F106	IBS mispredicted taken branch op	IbsOpBrnRet & IbsOpBrnTaken & IbsOpBrnMisp
F107	IBS return op	IbsOpReturn
F108	IBS mispredicted return op	IbsOpReturn & IbsOpMispReturn
F109	IBS resync op	IbsOpBrnResync

All events in Table 32 (F103-F109) are applicable only for ops that follow AMD64 branch semantics and therefore does not include micro-code branches.

G.5.5 Derived events for IBS load/store operations

Table 33 summarizes derived events for ops that perform load and/or store operations. With the exception of the first three derived events, the condition (IbsLdOp | IbsStOp) is assumed.

Table 33. Derived Events for Ops That Perform Load and/or Store Operations

ID	Name	Derivation
F200	IBS All Load/Store Ops	IbsLdOp \| IbsStOp
F201	IBS Load Ops	IbsLdOp
F202	IBS Store Ops	IbsStOp
F203	IBS L1 DTLB Hit	~IbsDcL1tlbMiss & IbsDcLinAddrValid
F204	IBS L1 DTLB Miss L2 DTLB Hit	IbsDcL1tlbMiss & ~IbsDcL2tlbMiss
F205	IBS L1 DTLB Miss L2 DTLB Miss	IbsDcL1tlbMiss & IbsDcL2tlbMiss
F206	IBS DC Miss	IbsDcMiss
F207	IBS DC Hit	~IbsDcMiss
F208	IBS Misaligned Access	IbsDcMisAcc
F209	IBS Bank Conflict On Load Op	IbsDcLdBnkCon
F20A	IBS Bank Conflict On Store Op	IbsDcStBnkCon
F20B	IBS Store to Load Forwarded	IbsDcStToLdFwd
F20C	IBSStore to Load Forwarding Cancelled	IbsDcStToLdCan
F20D	IBS UC memory access	IbsDcUcMemAcc
F20E	IBS WC memory access	IbsDcWcMemAcc
F20F	IBS locked operation	IbsDcLockedOp
F210	IBS MAB hit	IbsDcMabHit
F211	IBS L1 DTLB 4K page	~IbsDcL1tlbHit2M & ~IbsDcL1tlbHit1G & IbsDcLinAddrValid
F212	IBS L1 DTLB 2M page	IbsDcL1tlbHit2M & IbsDcLinAddrValid
F213	IBS L1 DTLB 1G page	IbsDcL1tlbHit1G & IbsDcLinAddrValid
F214	Reserved	
F215	IBS L2 DTLB 4K page	~IbsDcL2tlbMiss & IbsDcL1tlbMiss & ~IbsDcL2tlbHit2M & IbsDcLinAddrValid
F216	IBS L2 DTLB 2M page	~IbsDcL2tlbMiss & IbsDcL1tlbMiss & IbsDcL2tlbHit2M & IbsDcLinAddrValid
F217	Reserved	
F218	Reserved	
F219	IBS DC miss load latency	IbsDcMissLat when IbsLdOp & IbsDcMiss

The **IBS all load/store ops** derived event is a count of all IBS op samples that involve either a load and/or store operation. The **IBS Load Ops** and **IBS Store Ops** events break out the number of load and store operations performed by all sampled ops.

The **IBS WC Memory Access** derived event includes streaming stores to WB memory.

Detection of L1 and L2 DTLB miss events are more easily decoded than the similar events in IBS fetch samples, since all sampled ops are retired ops. Retired ops do not include speculative activity. All address translations must eventually complete in the L1 DTLB. Thus, the IBS translation page size flags for the L1 DTLB are always set according to the size of the completed translation. A 4K page L1 DTLB translation occurrs when both the IbsDcL1tlbHit2M and IbsDcL1tlbHit1G bits are clear. The L2 DTLB translation page size is valid when IbsDcL1tlbMiss is set. A 4K page L2 DTLB translation occurrs when both the IbsDcL2tlbMiss and IbsDcL2tlbHit2M are clear. Checking the IbsDcLinAddrValid bit is necessary to be sure that an address translation was attempted.

The **IBS DC miss load latency** is only valid for load operations. Miss load latency should only be tallied by software when the IbsLdOp bit is set.

The remaining derived events are simply the number of IBS op samples for which an event bit is set (or clear), e.g., **IBS MAB hit** is tallied when the IbsDcMabHit bit is set.

G.6 Derived Events for Northbridge Activity

The IBS Northbridge derived events measure the number of local and remote accesses (without regard to data source) and measure the number of local and remote accesses by data source. The event derivations in Table 34 assume that the overall Northbridge IBS data validity condition:

IbsLdOp & IbsDcMiss & (NbIbsReqSrc = 0)

is true.

Table 34. IBS Northbridge Derived Events

ID	Name	Derivation
F240	IBS NB local	~NbIbsReqDstProc
F241	IBS NB remote	NbIbsReqDstProc
F242	IBS NB local L3	NbIbsReqSrc=0x1 & ~NbIbsReqDstProc
F243	IBS NB local L1/L2 (inter-core)	NbIbsReqSrc=0x2 & ~NbIbsReqDstProc
F244	IBS NB remote L1/L2/L3 cache	NbIbsReqSrc=0x2 & NbIbsReqDstProc
F245	IBS NB local DRAM	NbIbsReqSrc=0x3 & ~NbIbsReqDstProc
F246	IBS NB remote DRAM	NbIbsReqSrc=0x3 & NbIbsReqDstProc
F247	IBS NB local other	NbIbsReqSrc=0x7 & ~NbIbsReqDstProc
F248	IBS NB remote other	NbIbsReqSrc=0x7 & NbIbsReqDstProc
F249	IBS NB cache M state	NbIbsReqSrc=0x2 & ~NbIbsReqCacheHitSt
F24A	IBS NB cache O state	NbIbsReqSrc=0x2 & NbIbsReqCacheHitSt
F24B	IBS NB local latency	IbsDcMissLat when ~NbIbsReqDstProc
F24C	IBS NB remote latency	IbsDcMissLat when NbIbsReqDstProc

These derived events correspond to the seven kinds of Northbridge activity described in Section 4.5.

Index

A

address-generation interlocks 115
arrays 14

B

base address register 300
boolean operators 18
branch prediction storage 100
branch target buffer (BTB) 229
branches
 based on comparisons between floats 38
 compound branch conditions 19
 dependent on random data 101
 optimizing density of 99
 prediction 229
 replace with computation in 3DNow! code 104

C

C language 19
 array notation versus pointers 14
 structures 31
cache
 64-byte cache line 91
caches
 probing 293
CALL and RETURN instructions 103
CI configuration
 data 297
code padding using neutral code fillers 68
code segment (CS) base, nonzero 104
Configuration Address Register (0CF8h) 297
Configuration Data Register (0CFCh) 298
const type qualifier 26
CPUID 201

D

data cache 196, 228
data organization 195
data-parallel threading 195
decoding 230
device NUMA 293
device's bus-number 300
DirectPath
 DirectPath over VectorPath instructions 52
displacements, 8-bit sign-extended 68
division 121, 122, 123, 124, 138
 replace division with multiplication, integer 32, 121
dynamic memory allocation consideration 21

E

extended-precision data 181

F

false data sharing 197
far control-transfer instructions 106
floating-point
 compare instructions 178
 division and square roots 36
 execution unit 234
 scheduler 233
 variables and expressions are type float 14
FXCH instruction 178

H

HWCR 305
HyperTransport assist 237
HyperTransport links 299
HyperTransport™ 293

I

I/O Thread Scheduling 296
if statement 20, 28
immediates, 8-bit sign-extended 67
IMUL instruction 125
index and data ports 298
inline functions 113, 130
inline REP string with low counts 128
instruction
 cache 228
 control unit 230
 short encodings 58
integer
 arithmetic, 64-bit 130
 division 32
 execution unit 231
 operand, consider sign 35
 scheduler 231
 use 32-bit data types for integer code 34
IO-device consumption 296

L

L2 cache controller 228
LEA instruction 56, 66

LEAVE instruction 64
load/store 23, 234
load-execute instructions 53
 floating-point instructions 54, 55
 integer instructions 53
local functions 28
local variables 33
LOCK 201
locked instructions 202
LOOP instruction 106
loops
 generic loop hoisting 26
 minimize pointer arithmetic 116
 partial loop unrolling 111
 REP string with low variable counts 129
 unroll small loops 17
 unrolling loops 110

M

memory
 dynamic memory allocation 21
 pushing memory data 118
memory barriers 200
memory-mapped I/O (MMIO) 294
MFENCE 201, 306
MMIO base and limit address registers 300
MMX instructions
 PREFETCHNTA/T0/T1/T2 instructions 84
Move/Compute 170
MOVZX and MOVSX instructions 116
multiplication
 by constant 125
 multiplies over division, floating-point 173
multi-threading 194
muxing constructs 105

N

non-coherent I/O 293
Northbridge 293

O

operands
 largest possible operand size, repeated string 128

P

parallelism 29
PCI configuration 297
 address 297
pointers
 dereferenced arguments 33
 use array-style code instead 14
population-count function 144
prefetch
 determining distance 87
 multiple 85
PREFETCH and PREFETCHW instructions 81, 84, 86
processor
 microarchitecture 224
prototypes 25

R

RDTSC 305
RDTSCP 3, 305
register reads and writes, partial 60
REP prefix 128

S

Scalar Merge Optimization 172
scheduling 109
serializing instruction 305
SFENCE 203
SHLD instruction 66
SHR instruction 66
SSE 147
SSE2 147
stack
 alignment considerations 94
store-to-load forwarding 21, 23, 75, 77, 79
stream processing 199
string Instructions 128
string instructions 128
subexpressions, explicitly extract common 30
superscalar processor 225
switch statement 28

T

task decomposition 195
task-parallel threading 195
time stamp counter 3, 305
timestamp counters 305
TSC 3, 305

U

unit-stride access 82, 88, 89

V

virtualized environments 305

W

WB memory 200
write combining 89, 239, 240, 241

X

XCHG 202
XOR instruction 129